First Edition (November 1991)

References in this publication to IBM products, programs or services do not imply that IBM intends to make them available in all countries in which IBM operates.

Publications are not stocked at the address given below. Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

Address comments concerning the contents of this publication to IBM Corporation, Data Management Market Support, C61/010, 5600 Cottle Road, San Jose, California 95193. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

(C) Copyright International Business Machines Corporation, 1991

Preface

The information contained in this document has not been submitted to any formal IBM test and is distributed on an "as is" basis without any warranty either expressed or implied. This document is intended to be a general guide only. The use of any information in this guide or the implementation of any of these conversion techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

No rights in IBM patents, copyrights or other intellectual property rights are granted or implied in the publication of this document.

This document contains references to third party software vendors. These references are for information only and do not constitute a recommendation or endorsement of these third parties and of their capabilities or products by IBM.

Permission is granted to copy this document for your internal use. The IBM copyright notice must be reproduced at the beginning of any portion copied.

This DL/I - SQL/DS™ Migration and Coexistence Guide is one of a series. Related publications include the *DBMS CONVERSION GUIDE* publications for ADABAS®, CA-DATACOM/DB®, CA-IDMS/DB®, and Model204®

This document was created using IBM BookMaster®, program number 5688-015.

ACKNOWLEDGEMENTS

This DL/I - SQL/DS Migration and Coexistence Guide was prepared by IBM's Data Management Market Support department in San Jose, California. The project leader was Bena Luxton, Advisory Marketing Support Representative. The Guide could not have been written without the valuable help of a truly international team of IBM contributors, as follows:

Michael I. Ball, Advisory Systems Engineer, Services Offerings Group, Houston (TX) Trading Area

Frances M. Cackowski, Senior Systems Engineer, Central New Jersey Branch Office (5D5)

Don Cameron, Senior Marketing Support Representative, Data Management Market Support, San Jose, CA

Allan DeLoach, Senior Systems Engineer, Knoxville, TN

Tom E. Early, Senior Systems Engineer, US Marketing and Services, Portland, OR

Mary S. Eidson, Senior Systems Specialist, Chicago (IL) Industrial Sector

Thomas G. Mirande, Advisory Systems Engineer, Edison NJ New Jersey Process Branch Office (448)

Michel Roy, Senior Systems Engineer, IBM Canada, National Support Center (East), Montreal (QC), Canada

Albert H. Sadler, Senior Systems Engineer, Professional Services, Central Western New York Trading Area, Rochester, NY

Chris Tett, Senior Systems Engineer, Country Systems Development Services Centre, IBM United Kingdom Ltd

Each person above contributed one or more sections to the Guide from his or her own expertise and actual experience with DL/I and SQL/DS installations. We would also like to thank the many reviewers of the book for their valuable suggestions.

We wish to acknowledge the CIM Advantage Production Planning System (CIMAPPS - formerly known as COPICS) development team of IBM in Sindelfingen, Federal Republic of Germany, for their development of many techniques for database and application migration. Many of their ideas were incorporated or adapted for this Guide. Users of CIMAPPS will be interested in also obtaining the migration guide for that system, IBM publication number SE12-5000.

TRADEMARKS

The following terms, used in this publication, are registered trademarks of the IBM Corporation in the United States and/or other countries:

AIX	OS/2
AS/400	PS/2
DB2	SQL/400
MVS/ESA	Systems Application Architecture

The following terms, used in this publication, have been adopted by the IBM Corporation as trademarks in the United States and/or other countries:

Distributed Relational Database Architecture	S/390
DRDA	SAA
DXT	SQL/DS
ES/9000	VM/ESA
QMF	VSE/ESA

The following terms, used in this publication, are registered trademarks of other companies as follows:

ADABAS	Software AG.
CA-DATACOM/DB	Computer Associates.
CA-IDMS/DB	Computer Associates.
EXPLORE for SQL/DS	Goal Systems International, Inc.
Model204	Computer Corporation of America

The following terms, used in this publication, have been adopted by other companies as trademarks in the United States and/or other countries:

UNIX	American Telephone and Telegraph Corporation

The first use of the trademark terms is denoted in the text of this publication by the symbol '™' for a trademark and by the symbol '®' for a registered trademark.

Contents

Chapter 1. Project Overview 1
1.1 How to Use This Guide 2
1.2 Why Introduce Relational? 3
1.2.1 The Relational Model 3
1.2.2 Technology 5
1.2.3 Systems Application Architecture® 6
1.2.4 Distributed Data 7
1.2.5 Multi-Vendor Database Interoperability 7
1.2.6 End-User Support 8

Chapter 2. Migration and Coexistence Considerations 11
2.1 Migration and Coexistence Strategies 12
2.2 Migration Techniques 14
2.3 Migration Personnel 16
2.4 Functional Changes During Migration 19
2.5 Project Planning Considerations 19
2.5.1 Major Task Project Plan 20
2.6 Education 21
2.7 Hardware/Software Considerations 21

Chapter 3. DL/I Application Systems Inventory 25
3.1 Migration Planning Data 28
3.1.1 Inventory Tables Content 29
3.2 Migration Implementation and Tracking Data 37
3.2.1 Migration Tables Content 37
3.2.2 Program Inventory Tables Content 42

Chapter 4. Workload Analysis 45
4.1 Database Migration Sizing 45
4.2 Application Program Migration Sizing 48
4.2.1 Application Programming Rules Of Thumb 49
4.3 Special Programs and Procedures 51
4.3.1 Data Conversion 51
4.3.2 Coexistence Programs 51

Chapter 5. Database Design 53
5.1.1 One-For-One Approach 53
5.1.2 Redesign Approach 53
5.1.3 Coexistence Approach 54
5.1.4 Limited Redesign Approach 55
5.2 Database Design Philosophy 55
5.2.1 Design Information Sources 55
5.2.2 Design Sequence 56
5.2.3 Data Naming Considerations 57
5.3 Hierarchical-to-Relational Considerations 58
5.4 Logical Database Design 60
5.4.1 Designing Tables 61
5.4.2 Designing for Relational 61
5.5 Designing for Data Related Between DL/I and SQL/DS 71
5.5.1 Migration of Dependent Segments 71
5.5.2 Migration of Root Segments Only 72

5.6 Physical Database Design 72
5.6.1 Physical Table Definition Considerations 72
5.6.2 Table definition 72
5.6.3 Storage Pools, Databases, and DBSPACEs 73
5.6.4 Indexes 74
5.6.5 Concurrency 74
5.7 PSB Replacement 74
5.7.1 Views of Data 75
5.7.2 Access Authorization 75
5.8 Database Design Checklist 75

Chapter 6. Data Migration 77
6.1 DL/I Database Unload 77
6.1.1 Considerations 78
6.1.2 Logical Unload Utilities 79
6.1.3 Write a Data Conversion Program 79
6.1.4 DL/I Unload Utility File 81
6.1.5 Creation of SQL/DS Objects 82
6.2 Loading Tables 83
6.3 Data Conversion Validation 84

Chapter 7. Application Program Migration 85
7.1 Introduction to the Migration of Application Programs 85
7.2 Overview of the Process 85
7.2.1 Programs to Be Migrated 86
7.2.2 Analyzing Existing DL/I Application Programs 86
7.2.3 Replacement of DL/I Interface Specifications 88
7.2.4 Converting the DL/I Calls or DL/I Commands. 89
7.2.5 Error Handling 89
7.2.6 Date Handling 90
7.2.7 Other Considerations 90
7.2.8 Program Preparation and Execution (Testing) 91
7.3 Program Migration Strategy 91
7.3.1 Analysis of DL/I Calls or DL/I Commands 93
7.3.2 Changing DL/I Calls or DL/I Commands To SQL Statements 93
7.4 Program Migration: Detailed Recommendations 94
7.4.1 Navigation Commands 95
7.4.2 Data Modification Commands 117
7.4.3 Referential Integrity 123
7.4.4 Iterative (Set) Processing 125
7.4.5 Handling Special Cases 128
7.4.6 Column Selection vs. Segment Selection 131
7.4.7 Error Handling 131
7.4.8 Scrolling 133
7.5 COBOL Programs 134
7.5.1 Host Variables 134
7.5.2 Indicator Variables 136
7.5.3 Initialization of Fields (Defaults) 136
7.5.4 Field-Level Programming 136
7.6 Unit Testing 138
7.7 Program Migration Checklist 138

Chapter 8. System Testing 141
8.1 Testing Methodology 141
8.2 Performance 141

8.3 Testing Steps . . . 142
8.3.1 Review SQL/DS Database Design . . . 143
8.3.2 Set up and Run Test Schedule . . . 144
8.3.3 Review Equivalent Function Test Results . . . 146
8.3.4 Reviewing Performance Tests . . . 147
8.3.5 All OK? . . . 148
8.3.6 Change Database / Application Design . . . 149
8.3.7 Preparing for Cutover to Production . . . 150
8.4 Production Execution . . . 151

Chapter 9. DL/I Security Migration . . . 153
9.1 DL/I Security . . . 153
9.2 SQL/DS Security . . . 154
9.2.1 Other Aspects of SQL/DS Security . . . 156
9.3 Migrating Security . . . 157
9.3.1 Chart Present Security . . . 158
9.3.2 Determine Security for SQL/DS . . . 158
9.3.3 Implement Security . . . 160

Chapter 10. Coexistence Strategies . . . 161
10.1 Extract Data for Query, Decision Support Systems . . . 161
10.2 Programs Requiring Data Stored on Both Systems . . . 162
10.2.1 Programs Updating a Single DBMS . . . 162
10.2.2 Programs Updating Both DBMSs . . . 162
10.2.3 Parent-Child Relationships Split Between DL/1 and SQL/DS . . . 163
10.3 Synchronization of Data Between Systems . . . 164

Chapter 11. Operations . . . 167
11.1 Operational Considerations . . . 167
11.2 New Operational Items . . . 167
11.3 System Backup and Recovery . . . 168
11.4 Job Control and Processing . . . 169
11.4.1 CICS Start Up . . . 169
11.4.2 DL/I Batch . . . 169
11.4.3 DL/I "MPS" Batch . . . 169
11.4.4 Application Job Control Language (JCL) . . . 169
11.4.5 SQL/DS Utility Job Control Language (JCL) . . . 170
11.5 Utilities . . . 170
11.5.1 Database Utilities . . . 170
11.5.2 Data Handling Utilities . . . 171
11.5.3 Database Recovery Utilities . . . 172
11.5.4 Operator Commands . . . 172
11.5.5 Data Definition and Access . . . 172
11.5.6 Housekeeping . . . 173
11.5.7 Report Utilities . . . 173

Appendix A. Hierarchical Database Architecture . . . 175

Appendix B. WITT . . . 179

Appendix C. Sources of Assistance . . . 181
C.1 IBM . . . 183
C.2 Andersen Consulting . . . 185
C.3 Carleton Corporation . . . 187
C.4 Computer Task Group . . . 189

C.5 GE Consulting Services 191

Glossary of Terms 193
General Definitions 193
SQL/DS Terms and DL/I Correspondence 193
DL/I Terms and SQL/DS Correspondence 195

Index 199

Figures

1. Inventory Table Relationships 28
2. Migration Table Relationships 37
3. Steps in SQL/DS program preparation and execution. 91
4. Example Structures Used in Describing the Conversion of DL/I to SQL. 106
5. Positioning Calls in a DL/I Structure 115
6. SQL/DS Table with Foreign Keys 116
7. Comparison of DL/I Status Codes and SQLCODEs (partial list) 132
8. Comparison of DL/I and SQL/DS IOAREA's 135
9. Sample SQL SELECT Statements. 137
10. Testing steps 143
11. Process To Compare Database Contents 147
12. Outline of DL/I security 154
13. Outline of SQL/DS security 156
14. Steps in security migration planning 157
15. Logical database structure (hierarchy) 176

Tables

1. DL/I PSB Table 29
2. DL/I PCB Table 29
3. DL/I Sensitive Segment Table 30
4. DL/I Sensitive Field Table 31
5. DL/I Virtual Field Table 32
6. DL/I DBD Table 33
7. DL/I Segment Table 33
8. DL/I Field Table 35
9. SQL/DS Tables 38
10. SQL/DS Tables 38
11. SQL/DS Column Table 39
12. SQL/DS Foreign Key Relationship Tracking Table 40
13. Field to Column Table 41
14. Program Table 42
15. Program to PSB Table 44
16. Relational Database Terms and Name Structures. 58
17. Delete rule correlation 68
18. DL/I Utilities 171
19. DL/I Utilities 171
20. DBSU facilities 171

Chapter 1. Project Overview

This document addresses the coexistence and migration of applications implemented with IBM's DL/I DOS/VS hierarchic data management system (DBMS) and IBM's Structured Query Language/Data System (SQL/DS™) relational DBMS. Many SQL/DS systems today are in DL/I installations, and it is expected that in a few years most of these DL/I installations will also have SQL/DS. Therefore, many VM and VSE installations in the 1990's will have both of these database managers.

DL/I is a hierarchical database manager with a comprehensive set of functions. The application programming interface to DL/I is known as *Data Language/One*, normally written *DL/I.* Today, it is being used by many installations for transaction systems.

SQL/DS was introduced in the 1980's and is a DBMS targeted for both operational and decision support applications. The productivity associated with the relational technology provided by SQL/DS has led many DL/I installations to implement SQL/DS. These organizations are interested in using SQL/DS for both informational applications and operational applications.

This document is intended to provide direction for both coexistence and migration so as to enable a productive DL/I and SQL/DS environment with a minimum of risk and expense, while maximizing benefits. This guide assumes that the transaction manager being used with DL/I will not change with the implementation of SQL/DS.

While keeping data and applications in DL/I, it may be desirable to have the same data available in SQL/DS and use the available tools to ensure that updates are synchronized. See "Migration Strategies" in Chapter 2 and Chapter 10 which addresses coexistence.

The term migration as used in this guide means migrating the existing DL/I application(s) to SQL/DS with minimal redesign of application logic. It is not meant to describe application re-engineering, where applications are completely redesigned and rewritten. The goal is to achieve the benefits of SQL/DS with minimum cost and in a reasonable time frame. The guide discusses the organization required for a coexistence and migration effort, the major tasks to be done, and tools and migration aids that can help in making a coexistence and migration successful.

Migration efforts may be easier and require fewer resources than new application development efforts. New development always requires significant resources devoted to requirements definition, design, coding and testing. Migration of an existing application requires significantly fewer resources applied to the requirements and design steps. Coding should not require as many resources as new application development, as most of the logic can be preserved. Testing, while it remains a major effort, should require less effort than testing for new development. Existing test cases can be used and expected results are documented. The guide includes suggestions to minimize this effort without reducing the validity of the results.

Some prerequisites to coexistence and migration are:

- Management commitment to the implementation of SQL/DS and the coexistence and migration effort.
- SQL/DS has completed installation verification.
- Standards and guidelines for the SQL/DS environment are in place.

The major topics discussed in the guide are:

- Coexistence/migration project management
- Migration personnel requirements and education
- Selection of application(s) for migration
- Workload analysis
- Database design
- Database migration
- Application program migration
- Testing
- Security, utilities and operations
- Database coexistence.

1.1 How to Use This Guide

This guide is intended for a number of different audiences and for different uses. It is intended to serve both as a tutorial on the process of migrating from an DL/I database to SQL/DS and as a reference for the coexistence and migration team.

For the DL/I user considering making a migration and concerned with the effort, this guide may be read in its entirety. For most this will be more detail than is required. Recommended reading would be the introduction and planning sections up through Chapter 4, "Workload Analysis" on page 45. The implementation sections that follow are more appropriate to data processing professionals.

For a complete consideration of the effort required, a database administrator (DBA) should read Chapter 5, "Database Design" on page 53 and an application analyst familiar with the programming standards of the installation should read Chapter 7, "Application Program Migration" on page 85.

Coexistence and Migration Project Manager: The individual doing the planning of the project should read the planning part of the book through Chapter 4, "Workload Analysis" on page 45 and may need to skim the balance of the guide to obtain a sense of the skills required.

Migration Consultant or Team Leader: The consultant or team leader responsible for planning and managing the migration should read this guide in its entirety. The guide assumes that the consultant is familiar with both DL/I and SQL/DS terminology and concepts.

Migration Personnel: For those who are to perform the migration of the application, this guide can be used as a reference. Chapter 2, "Migration and Coexistence Considerations" on page 11 may be read as an overview of the project. Beyond that, the individuals involved should select those chapters specific to their responsibilities.

DL/I Only or SQL/DS Only Personnel: While this guide assumes that the reader is familiar with both DL/I and SQL/DS terminology, there will be personnel with knowledge of only one of the database systems. For these people, the section entitled "Glossary of Terms" on page 193 is provided and should be reviewed before attempting to read the implementation chapters. It provides a correspondence between the DL/I terminology and SQL/DS terminology.

1.2 Why Introduce Relational?

Migration of an application from one database manager to another is not necessarily a productive effort. There are, however, a number of advantages of relational database which justify the migration of existing applications with little or no change in function. Over the years, DL/I users have made a very significant investment in application programs. For business reasons, many of these applications need eventual redesigns. At that time, the choice is typically to do new application development and major rewrites with SQL/DS and permit existing applications to atrophy.

If there is business justification to do a redesign of an existing DL/I operational application, the selected DBMS should be capable of handling both operational and decision support applications. However, in many cases, there is no business need for redesign of existing applications, yet the need for making the data available to end users still exists. In this situation, data extraction using vendor tools or user-written programs provides the basis for coexistence of the two database managers.

Additional factors should be considered: the cost and effort of maintaining two database managers, maintaining skills in multiple database management systems, assigning system resources such as memory to multiple systems and maintaining currency of code on multiple systems. Data interchange and consistency between the two systems is also a consideration.

1.2.1 The Relational Model

The relational model supported by SQL/DS provides significant value for an application migrated to SQL/DS. The following are benefits that both add value and reduce cost for these applications. In addition, as application extensions are made and new applications designed, further benefits will be realized.

Conceptual Simplicity: Relational data is stored as tables. This simple concept allows data processing analysts to communicate readily with application users on issues such as data content and relationships. It can also reduce training costs of personnel who will be working with the model. These capabilities extend to end user support as described below.

Normalization: Normalization isolates data to the simplest common design and provides a mathematically proven technique for reducing redundancy and reducing the risk that information will be inadvertently lost due to the deletion of associated data.

Functionally Rich Language: The functional richness of Structured Query Language (SQL) for both programmers and end users facilitates application enhancement and reduces costs of maintenance. In many cases, additional function can be added to an application with less effort than with other data models and with no change to the database. This means that application exten-

sions which were not justifiable with the existing database may become justified with the relational implementation.

Some of the key functions found in a relational database not present in the hierarchic model are:

Set processing: Data manipulation statements operate on an entire set of rows rather than a single row at a time. The rows which meet the search criteria of an SQL SELECT statement are retrieved in the form of a table. Updates, deletions, and insertions may also operate on sets of rows.

Sequencing: Selected sets of rows and columns may be sequenced as the application requires without regard to the stored sequence of the data.

Join: A single SQL statement can operate on multiple tables. (SQL/DS can currently join a maximum of 16 tables in a single statement). Data can be correlated among the tables based on any data values in the tables with either a static (compiled) or dynamic (ad-hoc) request.

Union: Multiple tables of similar format can be treated as a single large table for the purpose of a single request.

Referential Integrity: Referential integrity ensures that data in related tables remains consistent. The database manager will ensure that no inserts, updates, or deletes of rows are made that are inconsistent with existing related data. As implemented by SQL/DS, referential integrity is declarative, and maintained in the system catalog. Since the relationships are not maintained with the physical data, adding new tables and new relationships does not require physical reorganizations. The combination of normalization and referential integrity reduces the requirement for data maintenance, ensures data consistency, eases application extension, and avoids loss of information.

Industry Direction: Relational database technology has been embraced by the data processing industry. Standards for SQL, the relational database language, have been defined by major standards organizations such as ANSI, ISO, and FIPS.

As of Version 3 Release 2, SQL/DS is compliant with FIPS 127-1, ISO 9075-1989, and ANSI X3.135-1989 and X3.168 1989. SQL/DS will flag features in application source programs that are extensions of the SQL-89 standard or are not part of the SAA™ syntax standard. Standards compliance enables installations to get more utilization and productivity from their application programs while ensuring they will get consistent information and results from the output of the programs.

New applications from the PC to the largest mainframe are being developed using the relational model. This trend has several impacts on an installation. As end users look for new applications on personal computers, they will often be using relational database. As departmental computers are installed, they will use relational data base. Migration of host applications to a relational database provides a common base and format that is familiar to users.

For data processing personnel, working with the most popular VM and VSE relational database ensures them of a valuable skill. New personnel will recognize the value of learning and using the most current data base technology.

Courses and seminars are readily available for acquisition and exchange of information. SQL/DS user groups exist in most major cities for the sharing of experiences and knowledge. User group meetings often include presentations by key SQL/DS planners and developers, and leading edge SQL/DS users. There are many active SQL/DS projects within the national IBM user groups, GUIDE and SHARE.

Design Productivity: The addition of new data or relationships need not impact existing applications. The relational database is more "forgiving" of design errors and omissions. The declarative implementation of referential integrity ensures the separation of the maintenance of the relationship from the actual data. Hence, designs may be implemented more quickly with less future planning required. As a result, application development may start sooner.

Extensions to existing applications, and new applications which share data with existing applications, need not require redesign of existing related or shared data.

Extensibility: The relational model permits the addition of new data into a table, or new tables into a system, without the requirement for reorganization of existing data. Applications can be readily extended without having database changes impact existing programs. Additional function that could not be justified in the past due to the impact on existing programs or data will be more feasible on migrated applications.

Programmer Productivity: The relational model removes a number of responsibilities from the programmer. There is no structure to be maintained and no paths to the data to be navigated. Access to the data is the responsibility of the database manager, not the programmer. Applications may be prototyped more readily due to the simplicity of the relational model and the excellent tools available.

1.2.2 Technology

In addition to the relational model, SQL/DS also utilizes current state of the art technologies both in software and hardware. Applications migrated to SQL/DS may benefit from these technologies.

Optimizer: In a relational database the programmers and the end users are isolated from the physical storage of the data. The system must be responsible for navigation (finding and selecting data for the application). In the case of SQL/DS, it is the optimizer component that is responsible for navigation on behalf of an individual request. This expert system analyzes the current structure of the data, available indexes, and statistics on data volumes and values, and then selects the least cost path for accessing the data. As the database or processing environment changes, applications will not be affected. The changes affect only the decisions made by the optimizer on how data is located.

VM/ESA™ Utilization: SQL/DS is able to exploit appropriate features of the VM/ESA. operating system. Specifically, the SQL/DS VM Data Spaces Support (VMDSS) feature allows SQL/DS to exploit the VM/ESA implementation of Data Spaces and capitalize on the strength of the paging subsystem, including I/O blocking, prefetching, and main and expanded storage management. The

exploitation of microcode and new hardware instructions on selected ES/9000™ processors further demonstrates the synergy of VM/ESA, and SQL/DS.

VSE/ESA™: SQL/DS is able to exploit appropriate features of the VSE/ESA™ operating system, such as additional real storage and dynamic partitions.

Operating Environments: In the VM environment, SQL/DS operates in its own virtual machine. It services the data needs of both non-interactive (for example, preprocessors) and interactive development environments.

VSE can run as a guest operating system under VM. In this case, both VM and VSE users and application programs can share a VM controlled SQL/DS database.

The full services of SQL/DS are also available in the native VSE environment. Guest sharing enables VSE users to exploit the performance improvement of VMDSS.

1.2.3 Systems Application Architecture®

Systems Application Architecture (SAA) provides a common set of facilities across the hardware architectures that support user functions in OS/2® on the PS/2®, OS/400® on AS/400®, and VM and MVS on a S/370 or S/390™ host. SQL has been designated as the SAA Common Programming Interface (CPI) for database. SQL/DS implements the SQL CPI in the VM or VSE environments.

Connectivity: SAA defines connectivity between the different system platforms. Users connected to one platform can communicate with another platform to access data.

Portability: Applications written using SAA SQL and an SAA language such as COBOL or the SAA 4GL Cross System Product can be ported from one environment to another. They may be developed in an interactive environment such as VM/ESA and executed in another environment such as MVS.

Applications migrated to SQL/DS can then be extended through development in the environment most appropriate for the development process.

Distributed Data: SAA has introduced Distributed Relational Data. Over time, a user will be able to access and update data on multiple systems with a single transaction without knowing where the data exists or the type of system on which it resides. Detailed specifications for support of this heterogeneous distribution of data are provided by SAA Distributed Relational Database Architecture (DRDA™), described in more detail in 1.2.5, "Multi-Vendor Database Interoperability" on page 7. SQL/DS will provide this capability for the VM platform.

Data migrated to SQL/DS will be able to participate in SAA distributed relational database. This means users on other systems and other architectures such as OS/2 EE will also be able to utilize the data.

1.2.4 Distributed Data

The relational model is ideally suited to data distribution. A primary concern of a distributed data environment is performance due to the slow speed of the communications media relative to a local channel. Specific relational capabilities that support distributed data are:

Select: The relational select operation permits the user to restrict the data to be transmitted to only those rows required. Extensive Boolean and comparative operators limit the data selected. Subselects to qualify the request based upon data in the same or other tables further reduce the amount of data to be transmitted.

Project: The project operation of relational data permits the restriction of columns transmitted to only those required by the application. Data not required is not transmitted, reducing communications requirements.

Join: Joins permit the user to have extensive inter-table manipulations done at the location of the data before transmitting the resultant matched data to the user. This may result in significant communications reduction as no unmatched data need be transmitted.

Set Operations: The fact that relational operations are performed on sets permits a distributed data system to respond to a single request with a number of rows of data in a single transmission. The reduction in the number of requests and the ability to transmit responses of blocks of data helps optimize the overall system. Updates to sets may be made without transmitting the rows to be updated (or deleted).

Referential Integrity: Users need not be aware of constraints on inserts, updates, and deletes for related tables in another system. The database will ensure that no inserts or deletes of rows are made that are inconsistent with related data.

In addition, as referential integrity maintains the parent/dependent relationships, it supports better performance by avoiding the additional application requests that would otherwise be necessary to maintain the relationships.

1.2.5 Multi-Vendor Database Interoperability

Heterogeneous distributed environments involving multiple vendor DBMSs are becoming commonplace. In those environments, the presence of SQL/DS can significantly leverage both the access to and the management of distributed data. Most prominent relational DBMS vendors have developed gateways to SQL/DS; these gateways transparently perform all the necessary conversions (SQL dialects, data type conversion, error code conversion), and thus shield users and applications from the complexity of a heterogeneous environment.

DRDA: IBM's Distributed Relational Database Architecture (DRDA), a part of Systems Application Architecture (SAA), is a published architecture allowing connectivity to and among relational database managers operating in unlike operating environments. This new architecture prescribes commands, data descriptors, data objects, communications areas and statements as building blocks to support distributed database. DB2®, SQL/DS (VM), SQL/400 and OS/2 Extended Services have announced the specific release levels that will support

DRDA. In addition, other DBMS vendors have expressed interest in this open architecture for database interoperability.

Workstation to SQL/DS Connectivity: Various connectivity products specifically developed for SQL/DS by third-party vendors allow users and applications to access and modify data in SQL/DS from all major workstation operating systems (DOS, OS/2, UNIX™, AIX®, Macintosh). These products implement the client/server concept, and greatly expand the business application spectrum by letting users easily combine the power of graphical user interfaces and other applications typically available on the workstations with the robustness of SQL/DS's management of data.

By migrating data from DL/I to SQL/DS, users can take advantage of the breadth of these products and applications.

1.2.6 End-User Support

The requirements specific to an end user were key when the relational model was developed. It was intended to be suited to the free form need of the business professional as well as the needs of the data processing professional.

Ad-Hoc Query: Migrating data to SQL/DS provides the capability of ad-hoc query. Users can formulate questions and submit them directly to the database. The simplicity of the relational concept makes this possible. Tools such as the Query Manager of OS/2 or Query Management Facility (QMF™) of VM provide a consistent view of the data from any platform. Data migrated to SQL/DS provides more query opportunities with very little programming effort.

End-User Access: Many of the requests in a typical application programming backlog are for additional queries and reports against existing data. Relational data base simplifies the user's understanding of the data. Migration of applications to SQL/DS will open opportunities to end users. They will be able to define their own reports and change them at will. Reports will be more valuable and users more satisfied.

IBM SAA LanguageAccess: IBM SAA LanguageAccess implements a natural language interface to relational data which uses artificial intelligence. This can make data available to a new group of end users. The management decision-making process can be expedited by allowing direct access to information by those who normally have to depend on data processing to create specific programs to satisfy end user requests.

Application Extension: Most existing applications have a backlog of changes waiting for implementation. By migrating the application to SQL/DS, the addition of new application and function is made much simpler. The cost of individual changes will be reduced and previously marginal changes may now be cost justified. As the additions can be made more rapidly with less impact on existing programs, it will be possible to satisfy more requests. All of this will increase the value of data processing to the enterprise.

Decision Support: The flexibility of relational database makes it highly suitable for decision support. Data that is migrated to SQL/DS becomes available for use through products such as QMF, Interactive SQL (ISQL - an application that is delivered with SQL/DS), Application System (AS), and the IBM Data Interpre-

tation System (DIS). These products have powerful decision support capabilities such as statistical analysis, reporting, and graphics.

Chapter 2. Migration and Coexistence Considerations

The objectives of migration and coexistence with DL/I and SQL/DS are to achieve maximum benefits from the combination of hierarchical and relational database management system (DBMS) capabilities, while minimizing costs. The techniques involve limited, but intelligent, migration of data, and an analytic migration of selected application programs.

Even when you have decided which single application system you wish to migrate, you need to assess the affect of that migration on other application systems. Where potentially several systems will need to be migrated, some work must be done to discover the level of integration, so that choices can be made as to what migration(s) are possible and desirable.

Most organizations do not have one all-inclusive database. Rather, they have a number of systems or suites of programs which they have developed over time and which relate to each other. These multiple systems have a certain amount of dependency on one another, and it is crucial to examine these dependencies at an early stage in order to gauge the effect of differing migration strategies.

One process may update or create a file which is later added to another database. A typical example might involve updates collected during the day and then added, via a batch run, to another database or system at night. Consider a Delivery system that uses data in common with a Warehouse system, and adds details of delivery items that directly relate to existing database records. Thus Delivery and Warehouse are integrated systems. Alternatively, there could be a warehouse database and a delivery application which reads details from the warehouse database, then using its own files, keeps track of deliveries in progress.

This sort of information is vital in planning a migration, therefore Chapter 3, "DL/I Application Systems Inventory" on page 25 discusses methods for data and application analysis and evaluation in detail.

While in many cases, DL/I segments may be directly migrated to SQL/DS tables, the DL/I data may require additional normalization to improve the design, reduce redundancy, and eliminate ambiguous or imprecisely defined portions of the database. Database administrators will become familiar with conditions that may require structural changes between the two systems. DBA's involved in migrating the design of the data should be familiar with the meaning of the data.

In migrating the programs, it should be possible to remove some code required by the hierarchic data model, particularly code pertaining to DL/I navigation. Some programmed search loops may be eliminated. Developers doing the migration will become aware of patterns of program statements that are unnecessary in a relational environment.

Once complete, a migrated application system will use the capabilities of relational database; the migration costs should be only a fraction of new system development costs, since the basic business function of the application is not being redesigned.

An initial step in any project is to assess and organize your resources. Various strategic and tactical alternatives related to migration and coexistence are presented next, to assist in project and resource planning. The following section describes suggested team resource requirements. The chapter concludes with a discussion of the major tasks to be done for successful migration and coexistence.

2.1 Migration and Coexistence Strategies

In planning for use of both DL/I and SQL/DS in your organization, there are several alternative approaches. These are not mutually exclusive. You may want to adopt one strategy for one of your applications (or locations, or subsidiaries, etc.) and a different strategy elsewhere.

You may choose to make no changes at this time for a group of applications and the data related thereto. That choice would certainly be the correct one at this time if there are plans to redesign and redevelop the application in the near future (e.g. within a year). The time of redesign and redevelopment is then the proper occasion to introduce new software technology, including relational database management.

Relational Data Extracted For Query And Reporting: Many users have justified the introduction of a relational database system simply on the basis of making data much more available to end users. The strategy is to periodically extract or summarize from existing "production" data that may be stored today in older, non-relational databases or file management systems that are difficult for end users to access. The advantage of this approach is that no changes need be made to existing programs or data, and there is little operational impact on today's information delivery systems. You should consider how current the extracted or summarized data must be, and how much data is actually extracted or summarized, to institute the required additional operational procedures to provide end user access. You will need to assess which tools should be used by end users, but since there are so many excellent tools today which support IBM's relational database management systems, this job normally should involve evaluation of existing products, rather than a costly development effort.

Use Relational Database For New Applications Only: Many organizations have chosen to adopt relational database technology so as to achieve higher application development productivity and easier access to data by decision makers and other end users. In some cases, however, where today's applications are efficiently delivering the required information, and where the need for end user access to existing (as opposed to new and additional) data is not of high priority, there is no perceived need to change existing applications. So these organizations decide that new applications, and their related data, will be developed using relational technology, but existing applications will remain unchanged, or nearly so, continuing to use existing database management facilities.

The advantage of this approach is that the impact to existing production systems is minimized, but new applications may benefit from relational technology both from a productivity and usability standpoint. The introduction of relational technology is carefully controlled. There is no requirement for "mass conversion" of many existing applications. As decisions are made to com-

Redesigned Database, Transparent Or Minimal Application Change: This approach is commonly used where, as is often the case, an important motivation for implementation of relational is for improved end user access. In this approach, DL/I segments are analyzed in terms of a logical data model. The entities, attributes, and relationships are defined as as to be represented in what is termed "Third Normal Form[1]."

With this approach, however, there is minimal change to the application programs. Some of the navigational processing may (in fact must) be eliminated, but there may be minimal change to the procedural portions of the code. Thus the full exploitation of set processing and predicate logic may not be initially implemented.

Tools are becoming available that support this approach, that go beyond a simple one-for-one conversion and allow for normalization of the data. This technique can be very satisfactory for many applications and usually is less costly than a manual analysis and re-engineering of the logic of each program. It is thus often a good compromise choice.

Redesigned Database And Re-Engineered Applications: Ideally, all data should be migrated to properly designed tables and every application should be redeveloped to take full advantage of the relational environment. This is not always possible given cost, time, and other resource constraints. But you should not rule out this alternative for selected application systems if:

- You also want to introduce other new software technologies for that application, such as use of a fourth generation language;
- The performance is very critical, and you want the application path to be as efficient as possible;
- You believe that major functional changes will be required to the application system[2] in the not too distant future.

This alternative would ultimately, though perhaps not immediately, be the first choice for those applications which are being completely redesigned and redeveloped (replaced), as stated above.

Invariant Elements: There are portions of the application code that will not have to be changed even if you decide to migrate some of your databases from DL/I to SQL/DS. There are other functions of your environment that do not lend themselves to migration to relational, and for which no technical motivation exists to consider such migration.

---

[1] First Normal Form means that repeating groups are eliminated and the repeating fields (columns) are defined as a separate table. Second and third Normal Form further eliminate redundancy and what are termed transitive dependencies. Third Normal Form is generally desirable as both the usability and maintainability of the data are improved.

[2] However, we recommend in that case that the migration to the relational environment be done *first,* with minimal change to the application during the migration effort. The benefits are (1) you are not changing the business function while also migrating data - thus you have a "controlled experiment" to make testing easier; (2) once you have migrated to the relational environment, the backlog of change requirements may be addressed and often accomplished faster than if the application(s) were still on the non-relational platform. There are cases where this may not be possible due to business requirements. In that situation, it is best to make the functional change in both the non-relational version and the relational version, if any part of it exists already, again to facilitate a controlled test process.

Some elements of your environment that should not change upon migration from DL/I to SQL/DS are:

- CICS environment - we do not recommend changing your transaction manager environment. CICS and SQL/DS are well-integrated products that were designed for coordinated recovery and transaction management. Screen definitions and application program interfaces associated with data communication and transaction management need not change.
- Other types of data - You should continue to use the facilities of CICS and VSE as before for data such as print spool and DFHCOMMAREA.
- Procedural code unrelated to database access - the portions of your application program that perform calculations, algorithms, business rules, print formatting, and other functions unrelated to database access, should not have to be changed. The only exception would be where you decided to take advantage of an SQL/DS function, such as DATE data type, that wasn't in DL/I but enhances your application.

2.3 Migration Personnel

During planning for a migration, a concern may arise regarding the potential loss of application development resources for the period of the migration. Developers involved in migrating applications are less available for new function development and end user organizations will be justifiably concerned if they perceive no improved function from data processing for an extended period.

For that reason, this guide is built in such a way that an outside organization, IBM or other consulting organization, may use it. Methodologies are suggested that do not require the involvement of the original developers of the application or even a great deal of knowledge of the application. Resources must be available to the migration team to interpret application requirements, but the majority of the migration team need not be intimately familiar with the implementation.

The team actually doing the migration, that is migrating lines of code and data to the SQL/DS format, may be a temporary organization. There are benefits to considering a firm specializing in migration for this effort. Since the skills of mapping the DL/I data and applications into the SQL/DS environment may not be required after the migration is complete, engaging persons already versed in such migrations may be more economical, especially on the first application(s) converted. Such skilled personnel can transfer their knowledge to in-house members of the migration team for future migrations. Also, with the use of outside services, the migration team may act relatively independently in migrating portions of the application system. Internal developers will be able to continue in other areas delivering new function. However, people from your organization who understand the applications and data should be available to assist and answer questions.

This section defines the duties of individual team members. It is intended to be an inclusive but minimal requirement list, as each migration situation differs. Any of the listed positions below could be filled either by internal personnel, or personnel from IBM Professional Services, or other migration specialist firms.

Team size will vary widely between installations and applications, depending on the size of the migration effort. In some cases, a one or two member team will supply all of the necessary skills while other migrations may require larger teams. Each job or task mentioned below does not necessarily equate to a person. In some cases, team members may be temporary or part time, as the case requires.

Project Planner/Manager: The project manager has responsibility for staffing, planning, organizing and controlling the overall migration project. This position should be staffed by a data processing manager with strong project management experience and a good understanding of the technical issues involved.

Team Leader: The team leader is the key to the successful migration of an application system. This person must have good project management and control skills. For the first migration effort, an experienced consultant is a good candidate to fill this role. An additional requirement is adeptness at skills transfer and training.

The team leader reports directly to the project manager for migration activity accountability. He or she provides current status for each program assigned to his or her team on a weekly basis. Project management tools should be utilized to track the team progress and to highlight any project bottlenecks. In a large migration effort, there may be a need for multiple teams, thus multiple team leaders.

The team leader will review each program to identify the non-standard changes that must be made. He or she will assign the tasks necessary to migrate the programs to the appropriate members of the migration team and track their progress on a continuing basis. Additionally, the team leader has overall responsibility for program testing and cutover.

A working knowledge of the transaction manager (CICS), DL/I, SQL/DS, an application language such as COBOL, and JCL is essential. An understanding of the system flow of the application being converted and its relationship to other applications in the environment is desirable. If the team leader does not possess the application knowledge, then unrestricted access to a knowledgeable resource is essential.

In the VM/VSE environment, a working knowledge of REXX will prove to be an asset.

DL/I DBA: The DL/I database administrator assigned to the project will be involved in all phases of the application system migration. The DL/I DBA may also be the SQL/DS DBA in some installations. The main tasks for this function are:

- Have a thorough knowledge of DL/I database technology
- Be able to provide authorization and access to data and file structures in the DL/I database
- Have knowledge of how to load and unload an DL/I database
- Provide for database integrity
- Coordinate the migration effort with SQL/DS DBA
- Be responsible for developing backup and recovery plans for the application and database
- Have a detailed understanding of DL/I monitoring and performance tuning.

SQL/DS DBA: This function is usually involved with most phases of application implementation and migration. The main tasks for this position are:

- Coordinate the migration effort with DL/I DBA
- Design and implement the SQL/DS tables
- Define and maintain the SQL/DS installation parameters
- Direct SQL/DS backup, recovery, data integrity and data security procedures
- Monitor the SQL/DS subsystem with systems monitoring tools
- Authorize access to data and distribute database authority
- Load the SQL/DS tables
- Provide for database integrity
- Audit the system
- Identify and resolve SQL/DS performance issues.

Systems Programmer: The function of the systems programmer as a part of the migration team is to:

- Coordinate any system related tasks for the migration team
- Act as a consultant to the team for any systems related activities that may occur
- Consult on any CICS or VM related issues as they pertain to application migrations
- Be responsible for overall system tuning and software maintenance and installation.

Application Programmer: The application programmer will be given a specific assignment by the migration team leader. This may be several programs, one program or a portion of a large program from the application.

A working knowledge of DL/I and SQL/DS is essential. An understanding of the programming language being used is required, but the level of expertise will vary depending on the individual programmer assignments.

The application programmer:

- Makes source code changes where required
- Unit tests the new code
- Documents the program according to standards
- Records the completion of the program migration
- Aids the team leader in application system testing.

Business (Application) Analyst: This position requires an individual with extensive knowledge of the business side of the organization and who also has experience with the system being converted. The main tasks are:

- Serve as a business consultant to the team
- Be aware of the implementation of newly converted systems and the impact it will have on the business
- Act as an interface to internal users of systems for fact finding and trouble shooting
- Specify system test requirements
- Specify system acceptance procedures and criteria
- Help with resolution of data issues.

Operations: The operations specialist will be responsible for:

- Operational design and review
- Migration of all JCL or EXEC related components of the application system
- All JCL or EXECs for development, testing and production
- Migration of operational procedures
- System backup and recovery procedures.

2.4 Functional Changes During Migration

This guide follows the principle that functional changes should *not* be made during the migration. While one of the major reasons for the migration is to more readily accommodate changes to the relational system, these changes should not be implemented during the migration itself.

The effort to test the migration is a significant part of the overall effort. The effect of the addition of new function with changes to data, programs, inputs and outputs dramatically complicates the effort of verifying the migration. The easiest means of testing the converted system is to use identical test streams to both the former DL/I system and the new SQL/DS system, comparing the resultant databases and other outputs for equal results. As the results should be equal, it is possible to verify the results to a great extent using automated techniques.

As requirements for changes to the application are recognized during the migration, they should be logged for later implementation. Failure to follow this recommendation may extend the period of the migration and increase the cost.

2.5 Project Planning Considerations

Some level of planning must be done for the overall migration project. This planning encompasses education and environment requirements, the selected approach to migration staging, and the initial analysis of the applications under consideration.

The project chosen for the first migration effort should have a plan that addresses the major tasks and subtasks for analysis and implementation. Subsequent migration projects will have similar plans, refined to reflect actual experience.

The project plan can be finalized as soon as the workload analysis is completed. At a minimum, the plan should include measurable checkpoints, completion criteria, resource requirements, team structure, cost estimate and schedule completion date. There are many project planning tools available, such as Application System (AS) from IBM, that can be used.

The methodology used for cost estimating should take the following factors into account:

- Inventory of data and application programs
- Estimated complexity
- Estimated cost per unit
- Added hardware cost, both permanent and temporary
- Staff requirements.

The project should be tracked as closely as possible to provide feedback for measuring the accuracy of the initial project schedules and cost estimates. At the completion of the project, the actual cost can be used to validate and adjust the estimating techniques used.

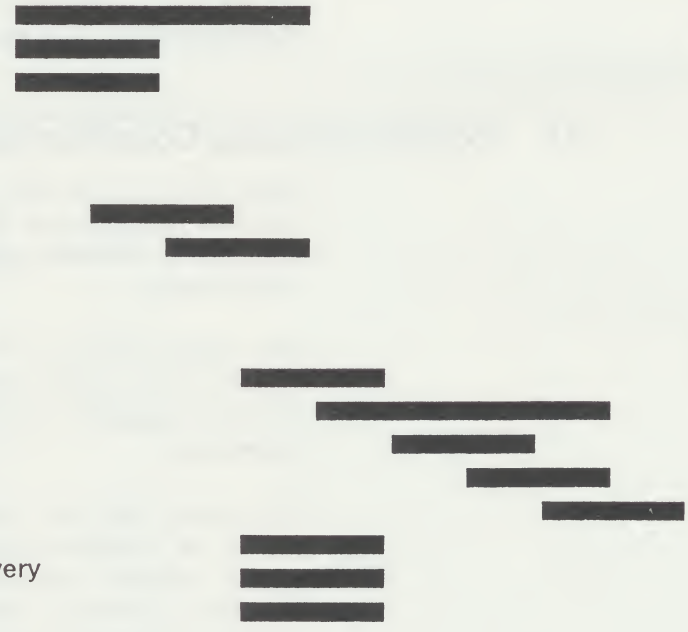
When using metrics such as lines of code, compare the lines of code workload estimate to the actual effort. This comparison should validate the program migration productivity. Note that the converted code may be less than the original code due to code removal for functions not required for SQL/DS implementation. Also, be sure to consider learning curves and orientation, as these are a one-time cost incurred during the first migration project.

2.5.1 Major Task Project Plan

The following time-lines show the relationship between the major tasks. Each time-line is intended to show when the task can be performed in relation to the others. The length of the time-line is an estimate for planning only. The actual time will depend on the size of the project.

The tasks and subtasks are not meant to be inclusive of every task (subtask) required for the migration. They cover the areas unique to the migration discussed in this guide. It is recommended that these tasks be used as input to the local project planning procedures and tools.

- Planning
  - Education
  - Environment
  - Application Selection
- Analysis
  - Detail Inventory
  - Workload Analysis
- Implementation
  - Database Design
  - Data Migration
  - Program Migration
  - Integration test
  - Production Cutover
  - Security
  - System Backup/Recovery
  - Operations



Post Project Review

- Pending Maintenance

  Evaluate any changes that were delayed until production cutover.

- Project Review

  Evaluate the entire project per local practices.

- Cost Validation

Evaluate the accuracy of the workload estimates and refine if necessary for the next migration phase.

2.6 Education

Education will be a key item in your successful migration from DL/I to SQL/DS. Education and understanding mean confidence on the part of the migration team. A realistic assessment of the skills of the migration team will give the project manager a base for education planning. Not all education requirements are necessarily met by formal classroom programs. Considerable skill transfer activity takes place in informal on-the-job training, meetings, judicious use of consultants and self-study.

It is strongly recommended that the project manager and team leaders complete the IBM Independent Study Courses listed below or have equivalent project management experience.

- Project Management WS165
- Managing Application Conversion Projects WS167.

The minimum education for all staff working on the migration is a working knowledge of the material contained in SQL/DS Fundamentals and SQL Workshop classes or their equivalents. IBM offers an extensive SQL/DS/QMF/Cross System Product curriculum. The courses are described in the *Catalog of IBM Education*, G320-1244.

When the decision has been made to use Cross System Product, there is a requirement for a Cross System Product coordinator to provide assistance in using the product, maintaining common routines and to assist in new programmer education.

It is advisable to have a Help Desk to provide end-user education and assistance. This position is usually staffed by at least one SQL/DS/QMF trained person.

To take advantage of the new relational capabilities provided by SQL/DS and QMF, it is suggested that some end users attend classes in QMF and SQL. These people can then take the lead in local or departmental training with assistance from the Help Desk support staff.

2.7 Hardware/Software Considerations

Processor Requirements: In general the CPU requirements for migrations are not significantly different from other development efforts. Actual requirements depend on the specific application.

The final testing and production cutover with both DL/I and SQL/DS running similar work loads will likely be the peak CPU load. This peak load may be estimated by summing the time for applications running concurrently on both systems.

DASD Requirements: Both database products, DL/I and SQL/DS, will exist in the system during the migration. Some portion of the data may be replicated for the purpose of development, test and production cutover. Most develop-

ment and test may be done with minimal size test data. During the initial phases the amount of test data may remain small. At the point of production cutover, the data used by an application is loaded into SQL/DS tables. After cutover verification, the DL/I version of the data can be archived, minimizing the time when duplicate data is online. Planning to migrate data and applications in phases rather than all at once may reduce DASD requirements.

The section on "Managing Database Storage" in the *SQL/DS System Administration* (VM: GH09-8084, VSE: GH09-8096) has recommendations for allocating SQL/DS related datasets among several volumes of DASD to reduce contention from other activity on the same I/O subsystem.

Software Requirements: This guide is focused on the migration to SQL/DS using Cross System Product and QMF as well as COBOL. It assumes that COBOL programs will use SQL calls. In addition, QMF may be used for many purposes, especially online queries and reports. QMF provides end users direct access to data rather than submitting program requests to data processing. Compile/QQF, a product marketed by PLATINUM *technology, inc.*, can be used to generate structured COBOL programs from stored QMF queries and report definitions. This product is available only in VM.

IBM offers many products useful in the migration process and ongoing operations. Additional products from IBM include:

- IBM SQL Master/VM - useful to automate many of the DBA functions (VM only).
- Data Base Edit (DBEDIT) - useful for data entry into a relational database, and for many simple full-screen applications (VM only).
- Workstation Interactive Test Tool (WITT) - Captures inputs and responses for later testing; can use a PS/2 to run regression testing of online applications.
- Teleprocessing Network Simulator (TPNS) - for performance/stress testing with a simulated network and terminals; workloads are recorded during production or test intervals to be subsequently rerun in various test environments. Variations to the original workloads may be made such as: number of terminals, workload data, workload programs, transaction arrival intervals and terminal input wait time.

Other vendors' products are also available to assist with various aspects of the migration effort. Some program packages are:

- Bachman Information Systems Inc., Burlington, MA: Re-engineers data from existing DL/I databases, normalizes the data, and produces a DB2 database design. While this product is designed today for DB2, the design information can be used with SQL/DS. Some of the physical constructs (for example, free space assumptions) will need to be modified. Bachman plans to deliver SAA data base design products for SQL/DS, SQL/400, and OS/2 Extended Edition Database Manager.
- BGS Systems, Waltham, MA: VM Application Planner does resource usage analysis before the coding stage.
- Carleton Corp., Burlington, MA: CQS-Data Engineer extracts data from DL/I to an SQL/DS loadable format.
- Compuware, Farmington, MI: CICS Playback will capture data for testing, allow unattended playback and compare the results. Comparison can be

made on screens, messages and reports. The comparison can be in character, hexadecimal, or full screen formats.

- Goal Systems International, Inc. EXPLORE® for SQL/DS is a performance monitor for SQL/DS under VSE.
- VM Solutions, Pleasant Hill, CA:

  DBEDIT PLUS edits SQL/DS tables and allows users to issue CMS commands.

  SQL/RADIUS is a set of integrated utilities for object management, storage management and system monitoring.

  GPR (General Purpose Reformatter and File Generator) can be used to audit tables or create tables from flat files (VM).

- VM Systems Group, Inc., Vienna, VA:

  DB/CENTER is a facility for automating and scheduling data base activities such as log archiving and database reorganization. The product is the functional integration of three individual products, DB/MONITOR for monitoring, DB/REORGANIZER, and DB/ADMIN (VM).

  DB/REPORTER is a batch report writer (VM).

  DB/EDITOR is a full screen utility for browsing and updating tables (VM).

  SQL/MENU is a full-screen applications generator for end users and application developers (VM).

Please consult the third-party vendor for details concerning any particular product.

The *SQL/DS Applications, Tools, and Services* (GX09-1218) offers a more complete listing of tools applications, tools, and services.

In addition to the information contained in this chapter, three other sources are particularly instructive.

The *SQL/DS Database Administration* (VM: GH09-8083, VSE: GH09-8095) provides techniques for estimating SQL/DS requirements.

The *SQL/DS Application Programming* (VM: SH09-8086, VSE: SH09-8098) provides first time users of SQL/DS with pertinent information to set up a detailed plan for implementing SQL/DS applications.

The *Referential Integrity Usage Guide*, (GC24-3281) reviews the concepts of referential integrity as implemented in SQL/DS, and gives advice on how to design the database and applications for referential integrity.

Chapter 3. DL/I Application Systems Inventory

In order to prepare accurate estimates of workloads, costs and schedules, it is necessary to make a thorough inventory of each application system to be migrated. This chapter provides direction and guidelines for conducting an inventory of the DL/I applications, analyzing the results of the inventory and assessing each application in order of migration difficulty. The data that will be collected and the criteria that apply to that data will help make the final assessment and selection as objective as possible.

As stated above in Chapter 2, "Migration and Coexistence Considerations" on page 11, information on relationships among systems of applications is vital in planning a migration. These relationships may not be immediately evident. For example, two application systems may not use any of the same databases, but one may create a sequential file used by the other. If two application systems are in any way related, the following questions must be addressed:

- If so, should both application systems be migrated?
- Is it possible to migrate one separately?
- Should they be migrated separately or together?
- If both should be migrated, which should be done first?
- To what extent should they be integrated after migration?

If database data are shared, then you may need to consider coexistence of different types of data as discussed in Chapter 10, "Coexistence Strategies" on page 161.

Therefore, before any detailed work is started, to assess the impact of a migration it is necessary to produce:

- An inventory of application systems
- An inventory of main database or file groups (not at segment/entity level but corresponding to, e.g., a DL/I database)
- Cross reference list of major and minor users of data groups (which could be part of the data inventory).
- Table or diagram showing dependencies, links and data flows.

It is useful for later work to know at this stage the status of the application systems. This is useful in deciding how to approach migration of the system(s) chosen. If a system is new, minimum change is probably best. If an application system is old, and has many known problems, it may be better to redevelop it. Application systems could be classified as:

- New
- In enhancement mode
- In maintenance mode
- Stabilized
- Scrap (at end of useful life)
- Mixed (some programs in one state and some in another)

A similar analysis should be done for the data. If a database is newly designed, then it likely may be migrated with minimal disruption. If a system is old, has had many changes, and contains, for example, different data in the same segment differentiated by key range, then further data analysis is appropriate.

Without looking at details, consider the approximate status of the data:

- Third normal form (easy to migrate to SQL/DS)
- Based on entity modelling
- Ad-hoc - layout of files/segments to suit application
- 'Dirty data,' e.g. certain values have special meanings
- Data in programs - e.g. some data is hard coded to avoid I/O
- Mixture

This analysis should be at a high level to develop a strategy. Once you have information about applications and their data, you are in a position to select one part (application system and data) for the first migration, and to decide the basic approach for the data, the application system, and the coexistence strategy.

The DL/I DBD and PSB definitions will be of considerable assistance in this effort. DBD and PSB generation listings can be used for a great deal of this inventory.

An additional and required source of information for the application inventory and resulting analysis is a DL/I database administrator (DBA) who is knowledgeable in the DL/I database system and the applications.

The following pages contain detailed descriptions of databases and programs to be inventoried. We cannot overemphasize the importance of this inventory. *You should not attempt to do any steps of the migration without completing this inventory.*

The specific skills needed by participants in the migration project responsible for the inventory are:

- Strong DL/I knowledge, especially DBA information
- Ability to read and understand existing DL/I application programs

Basic information to be gathered and its use in the selection of an initial application for migration are:

- Application names that will be used for identification and tracking of the application.
- A brief business description of the application and its functions that will enable both application analysts and systems personnel to recognize it. If a corporate data model exists, the entities of the model should be identified to allow identification of overlapping function with other applications.
- The number of DL/I segments used by the application is an important determinant in relative complexity of applications. Typically the more segment types used, the more complex the migration.
- Isolation is a major factor in degree of complexity of an application and therefore its desirability as a migration candidate. A greater degree of sharing of data between applications complicates the migration. If applications can be identified which do not force data to be shared between DL/I and SQL/DS, less effort will be required.
- The number of programs for the application must be identified and recorded by language. The language will influence the migration effort for each program. Typically, programs written in assembler will be harder to migrate than those written in a high level language. Also, DL/I supports

applications written in RPG and SQL/DS in the VSE environment does not. Any DL/I application programs written in RPG must be identified as they will have to be rewritten in another language as part of the migration. (See 4.2.1, "Application Programming Rules Of Thumb" on page 49 in Chapter 4, "Workload Analysis" for further discussion of this subject.)

- As the application may be 'locked' against changes during the migration, the number and criticality of any outstanding change requests becomes important. Users with a large number of changes outstanding will tend to be intolerant of major effort being applied to their application without improved function. They will not understand that changes impede the migration. (On the other hand, if the changes are impeded by restrictions of the DL/I database, such an application may be a desirable initial choice.)
- Business timing of the migration may be a factor and the cyclical nature of an application should be recorded. For instance, an application used primarily at year end should not be completed in January where the results of the effort will not be available for nearly a year.

With this limited information gathered, it is possible to make a determination as to the candidates for migration with the highest priority and the lowest risk. While it is desirable to have a small, limited application for the initial phase to develop staff experience and confidence, business priorities may not permit this.

Before beginning the DL/I database inventory, if you are not familiar with DL/I, it will be helpful to understand the hierarchical database model. Appendix A, "Hierarchical Database Architecture" on page 175 describes the model and 5.3, "Hierarchical-to-Relational Considerations" on page 58 describes migration considerations between the models.

For each application, the following entities must be documented with the attributes that will affect their migration:

- DBD
- SEGM (segment)
- Field
- PSB
- PCB
- SENSEG
- SENFLD
- VIRFLD

Relationships between them must also be documented. This information will be necessary to initially determine the application complexity and amount of effort required to migrate. During migration, the inventory will support the database design and be used to track the completion.

The DL/I DBD and PSB generation listings are the primary source of segment and field inventory data. They contain key data definitions, index definitions, and segment relationships. In many cases you will also need to examine any copy books, etc. used by application programs for segment definitions, as not all fields may be defined in the DBDs. For CICS online programs, information about the programming language used and relationship between application programs and PSBs can be determined from the CICS PPT and the DL/I ACT.

Another possible source of inventory data is the logical database design documentation. This documentation should not be used unless there is some assurance that it is current.

3.1 Migration Planning Data

The design, development and maintenance of a system to inventory the DL/I entities as a base for the logical database design and to track the conversion to SQL/DS objects is an essential component of the migration effort.

This chapter defines a set of relational tables that can be used as models for documenting the application inventory and tracking the migration effort. These tables might be built in SQL/DS or the OS/2 Extended Services Database Manager. Actual use of a relational DBMS for the application and data inventory has several advantages:

- It is an easy way to get started and gain experience using relational technology with minimal risk.
- With ad-hoc query capability, it is easy to discover relationships among applications and data - some of which you may not have previously known.
- The information is useful in its own right as a sort of 'mini-dictionary' of current systems.

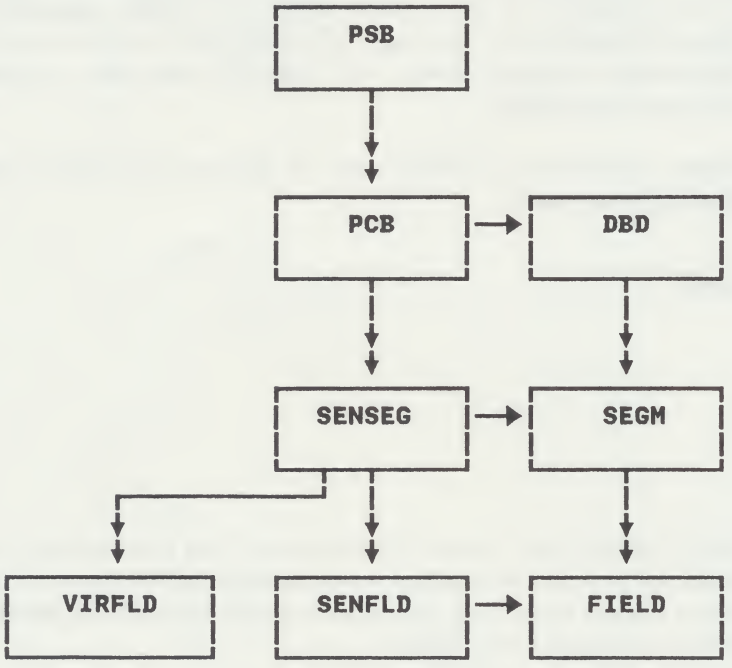


Figure 1. Inventory Table Relationships. This figure shows the relationships of the information used for the inventory to support planning and database design. Double arrows represent one to many relationships.

3.1.1 Inventory Tables Content

These data will have varied uses which are discussed in this guide. You may also identify additional information which would be unique to your installation.

The figure above shows those tables which are necessary to support the planning and initial design. Additional tables will be defined later in this chapter to provide documentation and tracking functions of the migration. The relationships indicated above are those that should be supported by referential integrity constraints to ensure the relationships are accurately reflected. Additional relationships may be exploited based on matching column values in the tables.

PSB Table

Column Name	Key type	Comments
PSB name	Primary	
Language		

Table 1. DL/I PSB Table. A table containing all DL/I PSBs

PSB Name: PSB name from the PSBNAME parameter of the PSBGEN statement.

This column should be defined as the primary key.

Language: The programming language of programs using this PSB, as defined by the LANG parameter of the PSBGEN statement.

PCB Table

Column Name	Key type	Comments
PSB name	Primary Foreign	Parent is defined in Table 1.
DBD name	Primary Foreign	Parent is defined in Table 6 on page 33.
PCB number	Primary	Number of PCB within the PSB.
Secondary index DBD	Foreign	Parent is defined in Table 6 on page 33.

Table 2. DL/I PCB Table. A table containing all DL/I PCBs

PSB name: PSB name from the PSBNAME parameter of the PSBGEN statement.

This column should be defined as the first part of the primary key.

This column should be defined as a foreign key with its parent the PSB Table.

DBD Name: DBD name from the DBDNAME parameter of the PCB statement.

This column should be defined as the second part of the primary key.

This column should be defined as a foreign key with its parent the DBD Table.

PCB Number: PCB number is the relative number of the PCB within the PSB.

This column should be defined as the third part of the primary key.

Secondary Index: DBD name from the PROCSEQ parameter of the PCB statement or NULL.

This column should be defined as a foreign key with its parent the DBD Table.

Sensitive Segment Table

Column Name	Key type	Comments
PSB name	Primary Foreign	Parent is defined in Table 1 on page 29.
DBD name	Primary Foreign	Parent is defined in Table 6 on page 33.
Sensitive segment name	Primary	
Parent segment name		"0" if none (i.e. for a root segment)

Table 3. DL/I Sensitive Segment Table. A table containing all sensitive segments defined in DL/I PCBs.

PSB name: PSB name from the PSBNAME parameter of the PSBGEN statement this SENSEG falls under.

This column should be defined as a foreign key with its parent the PSB Table.

DBD Name: DBD name from the DBDNAME parameter of the PCB statement this SENSEG falls under.

This column should be defined as a foreign key with its parent the DBD Table.

Sensitive Segment Name: Sensitive segment name from the NAME parameter of the SENSEG statement.

The combination of PSB Name, DBD Name, and Sensitive segment name can be defined as the primary key for this table.

Parent Segment Name: Segment name from the PARENT parameter of the SENSEG statement or the value "0".

Sensitive Field Table

Column Name	Key type	Comments
PSB name	Primary Foreign	Parent is defined in Table 3 on page 30.
DBD name	Primary Foreign	Parent is defined in Table 3 on page 30.
Sensitive segment name	Primary Foreign	Parent is defined in Table 3 on page 30.
Field name	Primary	
Field type		
Field length		in bytes
Exit routine name		blank if none

Table 4. DL/I Sensitive Field Table. A table containing all sensitive fields defined to DL/I.

PSB name: PSB name from the PSBNAME parameter of the PSBGEN statement this SENFLD falls under.

DBD Name: DBD name from the DBDNAME parameter of the PCB statement this SENFLD falls under.

Sensitive Segment Name: Sensitive segment name from the NAME parameter of the SENSEG statement this SENFLD falls under.

The combination of PSB Name, DBD Name, and Sensitive segment name can be defined as a foreign key with its parent the Sensitive Segment Table.

Field Name: Field name from the NAME parameter of the SENFLD statement.

The combination of PSB Name, DBD Name, Sensitive Segment Name, and Field Name can be defined as the primary key for this table.

Field Type: Record the data type of this field. You can use the indicator from the TYPE parameter of the SENFLD statement as a clue to the data type. However you should also check some application programs that use this field to make sure the DL/I specification, if any, is correct.

Field Length: The length in bytes of this field. This may be specified with the BYTES parameter of the SENFLD statement. However, DL/I does not require that this parameter be coded. If it is not specified you must determine the field length from the TYPE specification or by the way its corresponding field in the DBD is defined.

Exit Routine Name: The name of a user-written field exit routine if one was specified by the RTNAME parameter of the SENFLD statement, blanks otherwise.

Virtual Field Table

Column Name	Key type	Comments
PSB name	Primary Foreign	Parent is defined in Table 3 on page 30.
DBD name	Primary Foreign	Parent is defined in Table 3 on page 30.
Sensitive segment name	Primary Foreign	Parent is defined in Table 3 on page 30.
Virtual field name	Primary	
Field type		
Field length		in bytes
Exit routine name		blank if none
Field value		NULL if none specified

Table 5. DL/I Virtual Field Table. A table containing all virtual fields defined to DL/I.

PSB name: PSB name from the PSBNAME parameter of the PSBGEN statement this VIRFLD falls under.

DBD Name: DBD name from the DBDNAME parameter of the PCB statement this VIRFLD falls under.

Sensitive Segment Name: Sensitive segment name from the NAME parameter of the SENSEG statement this VIRFLD falls under.

The combination of PSB Name, DBD Name, and Sensitive segment name can be defined as a foreign key with its parent the Sensitive Segment Table.

Virtual Field Name: Virtual field name from the NAME parameter of the VIRFLD statement.

The combination of PSB Name, DBD Name, Sensitive segment name, and Virtual Field Name can be defined as the primary key for this table.

Field Type: Record the data type of this field. You can use the indicator from the TYPE parameter of the VIRFLD statement as a clue to the data type. However you should also check some application programs that use this field to make sure the DL/I specification, if any, is correct.

Field Length: The length in bytes of this field. This may be specified with the BYTES parameter of the VIRFLD statement. However, DL/I does not require that this parameter be coded for all data types. If it is not specified you must determine the field length from the TYPE specification.

Exit Routine Name: The name of a user-written field exit routine if one was specified by the RTNAME parameter of the VIRFLD statement, blanks otherwise.

Field Value: The initial value for this field, if specified by the VALUE parameter of the VIRFLD statement; NULL otherwise.

DBD Table

Column Name	Key type	Comments
DBD name	Primary	
DBD type		physical, logical, or index DBD

Table 6. DL/I DBD Table. A table containing all DL/I DBDs

Use the DBD generation listing as the source for this data. There should be one row in the table for each DL/I database in your installation including databases used as primary or secondary indexes and logical databases.

DBD Name: DBD name from the NAME parameter of the DBD statement this SEGM falls under.

This column should be defined as the primary key.

DBD Type: An indication of the type of DBD, i.e. this DBD defines a physical, logical, or index database.

Segment Table

Column Name	Key type	Comments
DBD name	Primary Foreign	Parent is defined in Table 6.
Segment name	Primary	
Parent segment name		"0" if root segment
Segment length		in bytes
Key length		fully concatenated key length
Duplicates		yes or no indicator
Segment type		physical child, real logical child, virtual logical child, or index segment
Physical parent DBD name		
Real logical child segment name		
Logical parent DBD name		
Logical parent segment name		

Table 7. DL/I Segment Table. A table of all DL/I segments and their characteristics.

DBD Name: DBD name from the NAME parameter of the DBD containing this segment.

This column should be defined as a foreign key with its parent the DBD Table.

Segment Name: Segment name from the NAME parameter of the SEGM statement.

The combination of the DBD Name and Segment Name columns should be defined as the primary key.

Parent Segment Name: Segment name from the PARENT parameter of the SEGM statement or the value "0" indicating a root segment.

Segment Length: The length of the segment as defined in the BYTES parameter of the SEGM statement. Use maximum length for variable-length segments.

Key Length: The length of the fully concatenated key for this segment.

Duplicates: Indicate if duplicate segments are allowed for this segment type. Segments defined without a unique sequence field or without a sequence field can have duplicates.

Segment Type: Indicate whether this is a physical child segment, a real logical child segment, a virtual logical child segment, or an index segment.

Physical Parent DBD Name: If this is a virtual logical child segment, this column should contain the name of the DBD where the real logical child segment exists. This will be the DBD name found in the SOURCE parameter of the SEGM statement that defines the virtual logical child segment.

Real Logical Child Segment Name: If this is a virtual logical child segment, this column should contain the name of the real logical child segment. This will be the SEGM name found in the SOURCE parameter of the SEGM statement that defines the virtual logical child segment.

Logical Parent DBD Name: If this is a real logical child segment, this column should contain the name of the DBD where the logical parent segment exists. This will be the second DBD name found in the PARENT parameter of the SEGM statement that defines the real logical child segment.

Logical Parent Segment Name: If this is a real logical child segment, this column should contain the name of the logical parent segment. This will be the second segment name found in the PARENT parameter of the SEGM statement that defines the real logical child segment.

Field Table

Column Name	Key type	Comments
DBD name	Primary Foreign	Parent is defined in Table 7 on page 33
Segment name	Primary Foreign	Parent is defined in Table 7 on page 33
Field name	Primary	
Sequence field		
Field type		
Repeating field		Fixed, Occurs, or Occurs Depending On
Field length		in bytes
SQL/DS column name		Initially NULL

Table 8. DL/I Field Table. This is an inventory of all the DL/I fields and their relationships to DL/I segments.

Often all fields in the segments are not defined in the DBD. A comparison between the application program segment definitions and the DBD SEGM and FIELD statements can assist in recognizing undefined fields. For those cases where parts of records are undefined, it will be necessary to determine the correct definition and add it to the inventory. Elements defined as "Filler" might appear to be null, but you should investigate to ensure that some application has not been using this space without definition.

The information in this table will be used to drive the logical design process as equivalent columns are defined.

DBD Name: DBD name from the NAME parameter of the DBD containing this field.

Segment Name: Segment name from the NAME parameter of the SEGM statement containing this field.

The combination of DBD Name and Segment Name should be defined as a foreign key with its parent the Segment Table.

Field Name: Field name from the NAME parameter of the FIELD statement.

The combination of the DBD Name, Segment Name, and Field Name columns should be defined as the primary key.

Sequence Field: Indicate if this field is used as a sequence field on the segment and if so, whether the sequence field is unique or not. This can be determined by checking for optional parameters coded after the field name in the NAME parameter of the FIELD statement.

Field Type: Record the data type of this field. You can use the indicator from the TYPE parameter of the FIELD statement as a clue to the data type. However you should also check some application programs that use this field to make sure the DL/I specification, if any, is correct.

Repeating Field: If this field is defined as a COBOL OCCURS or OCCURS DEPENDING ON field (or equivalent in other languages) in application programs indicate that in this column. This will be a flag to the design process that a non-relational condition exists that must be treated. The number of such entries is an indicator of migration complexity. For repeating fields, enter as "fixed", "occurs", or "occurs depending on".

Field Length: The length in bytes of this field. This may be specified with the BYTES parameter of the FIELD statement. However, DL/I does not require that this parameter be coded. If it is not specified you must determine the field length from the TYPE specification or by the way it is defined relative to other fields in the segment, or by the way it is defined in application programs.

SQL/DS Column Name: This data will be filled in during SQL/DS design. The column name entered here will be the one that are equivalent to the DL/I field names.

3.2 Migration Implementation and Tracking Data

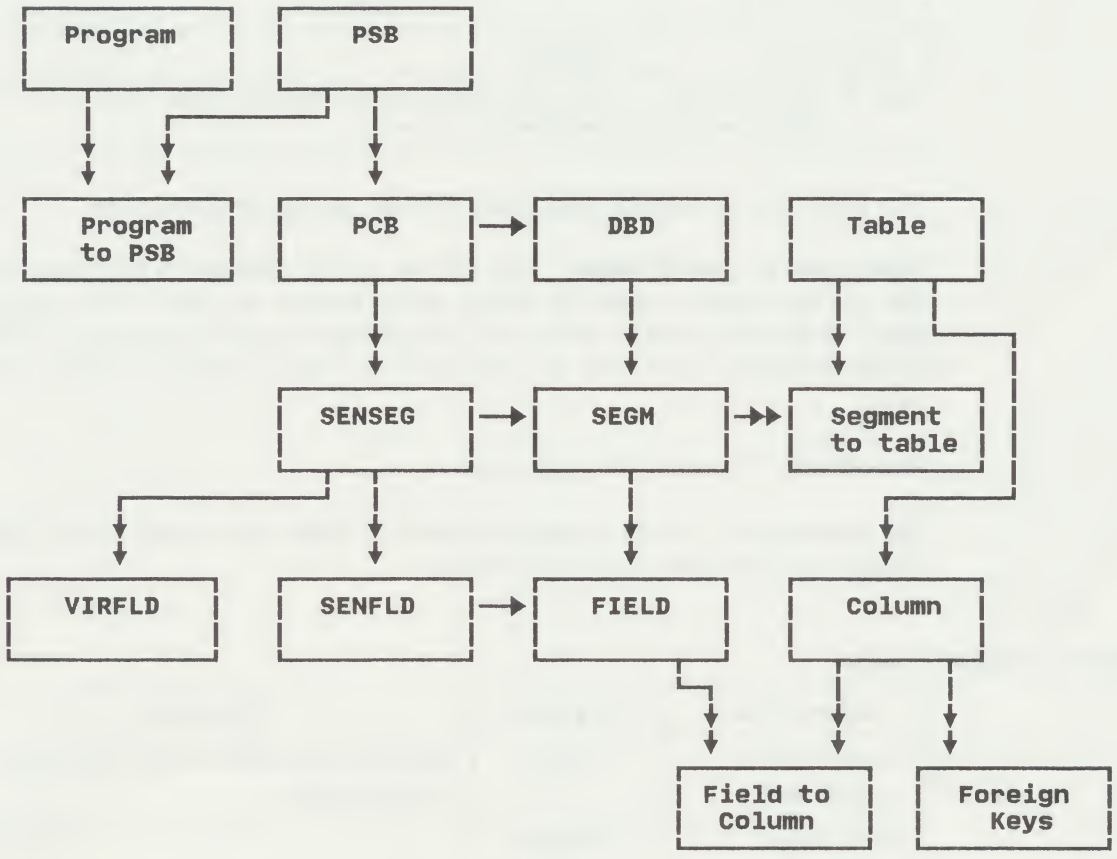


Figure 2. Migration Table Relationships. This figure shows the relationships of the tables used for both the inventory and for the implementation and tracking control. They consist of the inventory tables described previously, plus additional tables to reflect the SQL/DS database design for application program migration and to verify that all DL/I objects have been migrated.

3.2.1 Migration Tables Content

The figure above shows those tables which are necessary to support the documentation and tracking functions of the migration. Additional tables have been defined to track programs and SQL/DS tables and their relationships to the DL/I objects. The relationships indicated above are those that should be supported by referential integrity constraints to insure the relationships are accurately reflected. Additional relationships may be exploited based on matching column values in the tables. You may also identify additional information which would be unique to your installation.

These additional tables are now described.

'Table' Table

Column Name	Key type	Comments
Application or user-id name	Primary	Needed only if table name not unique in conversion.
Table name	Primary	

Table 9. SQL/DS Tables. This table identifies the SQL/DS tables that were created during the selected application conversion.

This table has an entry for each table defined during the design task.

Application or User-id Name: This column is only required if the table names will not be unique across all tables being created as part of this migration effort. If the table names will not be unique then it will be necessary to further qualify the table name with an application or user-id name to ensure uniqueness.

Table Name: The SQL/DS table name.

The combination of the Application/User-id Name and Table Name can be defined as the primary key for this table.

'Segment to Table' Table

Column Name	Key type	Comments
Application or user-id name	Foreign	Needed only if table name not unique in conversion.
Table name	Primary Foreign	
DBD Name	Primary Foreign	
Segment Name	Primary Foreign	

Table 10. SQL/DS Tables. This table cross references SQL/DS tables and the DL/I segments that are the sources for these tables.

This table is used to cross reference between SQL/DS tables and their source DL/I segments.

This table will be used during program migration to identify for the programmer the format of the new data in order that differences from the former DL/I segment can be accommodated.

Application or User-id Name: The application or user-id name necessary to uniquely identify this table.

Table Name: The SQL/DS table name.

The combination of the Application/User-id Name and Table Name can be

defined as a foreign key for this table with the 'Table' table as parent. The table name is also part of the primary key.

DBD Name: Name of the DBD that contains the segment that is a source for this table.

Segment Name: Name of the segment that is a source for this table.

The combination of the DBD Name and the Segment Name can be defined as a foreign key with the SEGM Table as parent. Both DBD Name and Segment Name are parts of the primary key.

Column Table

Column Name	Key type	Comments
Application or user-id name	Primary Foreign	Needed only if table name not unique in conversion. Parent is defined in Table 9 on page 38.
Table name	Primary Foreign	Parent is defined in Table 9 on page 38.
Column name	Primary	
Data type		
Field length		
Nulls		
Primary Key		If this column participates in a Primary Key, enter its relative column number within the key.
Foreign Key		Enter yes or no. If yes, provide information in Foreign Key Table.

Table 11. SQL/DS Column Table. This identifies SQL/DS columns defined in the migration and their attributes.

The Column Table describes both the redefined DL/I fields and the added information to support keys and other implied information. It will be the basis for application program migrators to understand the nature of the new data they are migrating to. Newly defined data such as keys may be identified by the absence of an entry in the "Field to Column Table". If desired, this information could also be reflected in a single character code column, i.e. a flag.

Application or User-id Name: The application or user-id name necessary to uniquely identify the SQL/DS table.

Table Name: The name of the SQL/DS table this column is defined in.

The combination of the Application/User-id Name and Table Name can be defined as a foreign key for this table, with its parent being the 'Table' Table.

Column Name: The catalog name of the column. Columns deriving from DL/I should have names close to, if not identical to, that of the former DL/I fields.

The combination of the Application/user-id Name, Table Name, and Column Name columns can be defined as the Primary Key for this table.

Data Type: This information will be necessary to reflect any differences in the DL/I definition of the field from that of SQL/DS. Record the data type as it is specified to SQL/DS.

Field Length: Record the length in logical characters or digits to enable comparison to DL/I lengths despite differences in specification techniques.

Nulls: Indicate whether nulls are allowed in this column. This information will be necessary for program migrators to determine whether there is any potential impact on the application programs.

Primary Key: This is primarily an indication that the column participates in a primary key. Application programmers will have to take this into account for inserts, deletes, and updates of this column.

Foreign Key: An indicator that this column participates in one or more foreign keys. The "Foreign Key Relationship" Table will further define the manner in which it participates. The programmer will need to take this information into account as rows are being inserted into the table containing this column.

Foreign Key Relationship Table

Column Name	Key type	Comments
Column name	Primary Foreign	Enter qualified column name. Parent is in Table 11 on page 39.
Relationship Name	Primary	Enter name of the relationship.
Key position		Relative column number in compound foreign key.
DL/I segment		Name of DL/I segment (if any) from which the key derives.

Table 12. SQL/DS Foreign Key Relationship Tracking Table. This provides a record of SQL/DS design decisions to replace segment relationships and relate data split by normalization.

This table identifies the characteristics of the foreign keys.

Column Name: The catalog name of the column. Columns deriving from DL/I should have names close to, if not identical to, that of the former DL/I fields.

This column should be defined as a foreign key with its parent the Column Table.

Relationship Name: Enter the SQL/DS relationship name. The name is required if the column is to participate in more than one foreign key. The information along with the name of the table and column identifies a use of the column within a specific foreign key. Columns used in multiple foreign keys will have additional entries in the table.

The combination of the Column Name and the Relationship Name columns should be defined as the primary key.

Key Position: If the column is used in a compound key, indicate which position this column holds within the key.

DL/I Segment: Name of the DL/I segment (if any) from which the column and relationship are derived.

Field to Column Table

Column Name	Key type	Comments
DBD name	Primary Foreign	Parent is defined in Table 8 on page 35
Segment name	Primary Foreign	Parent is defined in Table 8 on page 35
DL/I Field name	Primary Foreign	Parent is defined in Table 8 on page 35
Application or user-id name	Foreign	Needed only if table name not unique in conversion. Parent is defined in Table 11 on page 39.
Table name	Foreign	Parent is defined in Table 11 on page 39.
Column name	Foreign	Parent is defined in Table 11 on page 39.

Table 13. Field to Column Table. This table provides a cross reference between the DL/I fields migrated and their corresponding SQL/DS columns.

This table cross references the SQL/DS columns and the DL/I fields from which they are derived.

DBD Name: Name of the DBD that contains the segment where a related DL/I field occurs.

Segment Name: Name of the segment where a related DL/I field occurs.

Field Name: The name of the DL/I field that is a source for the SQL/DS column.

The combination of the DBD Name, Segment Name, and DL/I Field Name columns can be defined as a foreign key with the Field Table as the parent table. These coloumns also comprise the primary key for this table.

Application or User-id Name: The application or user-id name necessary to uniquely identify the SQL/DS table.

Table Name: The SQL/DS table name.

Column Name: The catalog name of the column.

The combination of the Application/User-id Name, Table Name, and Column

Name columns can be defined as a foreign key for this table with the Column Table as the parent.

3.2.2 Program Inventory Tables Content

In addition to gathering information about the data, it is also necessary to gather information about the programs to be migrated. The following section describes the data necessary to support the workload estimation and the actual migration process.

The first effort is to identify all of the programs used by the application and those programs of other applications that may share the data of the application.

Programmer-supplied lists of programs used in the application will provide a base point. They should not, however, be relied on as the complete set of programs. Exception programs and infrequently used reports tend to be forgotten even by the authors unless they were of particular note or recently developed.

Program Table

Column Name	Key type	Comments
DL/I program name	Primary	
SQL/DS program name		If SQL/DS program name different than DL/I.
Program description		
Program type		batch, batch MPS, or online (CICS)
Programming language		
Application use		Single or multiple.
Currency		Indication as to currency of program and source.
Change requests		Number of pending changes requested for this program.
Program size		Number of lines of code or local metric.
Number of DL/I calls		Number of DL/I call statements in this program.
Source code location		System, library, member name, etc.
Assigned to		
Code complete		Date
Test complete		Date

Table 14. Program Table. This table contains the names of all programs that are required for the migration of a selected application including migration programs.

This table contains the basic information about the programs needed to determine the workload of the migration and to support the migration personnel.

Programs can be related to the segments on which they operate through the Program to PSB Table and the PCB Table. Note that while the PSB is useful in determining this information, the PSB may define more segments than the program actually uses. A migration will appear more complicated than it is if segments are included that are not used by the program.

DL/I Program Name: The name of the source program so that developers may easily identify it. You may choose to record both a source name and an object name if the two are different.

This column should be defined as the primary key.

SQL/DS Program Name: If the SQL/DS name will be the same as the DL/I program name, and simply stored in independent libraries, this column is not necessary.

Program Description: Description of the major function of this program.

Program Type: Indicate whether the program is: batch, MPS batch, or online (CICS).

Programming Language: Identify the language used for this program. For online (CICS) programs you can use the CICS PPT to determine this. You can also check the LANG parameter of the PSBGEN statement of the associated PSB to determine the language used. Note that the LANG parameter may not accurately reflect the language of the application programs using this PSB however.

Application Use: Note whether the program is used in a single or in multiple applications. This information will help in determining the degree of interdependence of applications.

Currency: Indicate whether this program is currently in use and that the source reflects the state of the execution code. Programs may exist which are not currently being used and may no longer be correct. A decision will have to be made as to whether these programs should go through conversion.

Change requests: Record the approximate number of changes that have been requested for this program. If there is no direct log of requests, a degree indicator such as "heavy", "moderate", or "light" may be used. The intent is to identify the degree of user pressure for changes to this program.

Program Size: The generally accepted metric is source lines of code which will be used in the examples in this guide. Refer to 4.2, "Application Program Migration Sizing" on page 48 for further discussion of this topic. Local metrics which may be a better guide to program size and complexity (i.e., difficulty of conversion) may be used.

Number of DL/I calls: The number of DL/I calls in a program is a measure of the level of effort required to migrate that program to SQL/DS. Refer to 4.2, "Application Program Migration Sizing" on page 48 for further discussion of this topic.

Source Code Location: Record the location of the source code. The migration team will need to locate the source to convert it. In the remote possibility that the source does not exist, record that fact.

Assigned To: Record the programmer assigned to complete migration of this program.

Code Complete: Record the date the code changes are completed.

Test Complete: Record the date testing of the code is completed.

Program to PSB Table

Column Name	Key type	Comments
DL/I program name	Primary Foreign	Parent is defined in Table 14 on page 42.
PSB name	Primary Foreign	Parent is defined in Table 1 on page 29.

Table 15. Program to PSB Table. This table relates DL/I programs and their PSBs.

Because a PSB may be used by multiple programs and a program may use multiple PSBs, the Program to PSB Table is used to represent this many-to-many relationship. There will be a row in this table for each valid combination of program and PSB used in your installation.

DL/I Program Name: The name of the DL/I program.

This column should be defined as a foreign key with its parent the Program Table. It is also part of the primary key for this table.

PSB name: The name of the PSB the DL/I version of this program uses to communicate with DL/I. For online (CICS) application programs this can be taken from the PSBNAME parameter of the DLZACT TYPE=PROGRAM statement. For batch and MPS batch programs, this can be found on the batch parameter statement in the execution JCL.

This column should be defined as a foreign key with the PSB Table as its parent. It is also part of the primary key for this table.

Chapter 4. Workload Analysis

The objective of workload analysis is to size the database and program migration effort. The results are used to formulate migration cost estimates and produce the project schedule for the implementation phases.

The workload analysis methods outlined in this chapter will build upon the project plan outline and the results of the detailed application inventory. See 2.5, "Project Planning Considerations" on page 19 and Chapter 3, "DL/I Application Systems Inventory" on page 25 for the information necessary to complete these prerequisite tasks. Accurate estimates for workloads, costs and schedules necessitate a thorough inventory of the application system to be migrated. This inventory must include the DL/I databases, programs, and files used by the application being migrated. It must also include any application and data cross-reference relationships which might not normally be considered part of the application being migrated. An example of this consideration is an Inventory application that also updates or uses DL/I databases (or segments of databases) which are normally considered part of the the Accounts Receivable application.

The following sections present some guidelines for determining the level of effort required to migrate an application. This workload analysis is done separately for database design, application program migration and special purpose programs for data conversion and coexistence.

There may be operational requirements that require workload estimates in order to produce a complete assessment of the effort involved in migration. Examples are:

- Developing testing aids
- Developing separate SQL/DS tables used for application testing
- Obtaining and loading test data into these tables
- Developing security procedures
- Creating SQL data control language to implement the security plan
- Developing backup and recovery procedures
- Creating JCL to execute various SQL/DS utilities.

These topics are not addressed in this chapter. The &db2prim. discusses most of the above topics.

The guidelines that are provided in the following sections are rules of thumb for the first migration project. More precise estimating methods can be developed for subsequent projects based upon the results of the initial application. (This assumes that accurate records are kept on the effort expended on the development and implementation of the initial application.)

4.1 Database Migration Sizing

The sources of data needed for the analysis will depend on the way DL/I databases have been managed in the organization. The most desirable source of this information is a data dictionary which is up-to-date and accurate and contains all the necessary information. Reports from this data dictionary would provide a complete application and DL/I database inventory with all the neces-

sary information for analyzing the database design workload. However, many installations do not have such a data dictionary.

The topic of DL/I database inventory is discussed in detail in Chapter 3, "DL/I Application Systems Inventory" on page 25. Before beginning the analysis of the database design workload it is important to complete the inventory tables described in that chapter. The reason this inventory is so important is that it will provide the basis for an educated estimate for the number of SQL/DS tables which are needed for the application system being migrated from DL/I to SQL/DS. The number, size, and complexity of the tables will determine the amount of effort required for database design in the SQL/DS system.

Table Design Rules Of Thumb

Based on the considerations listed below, an approximate workload estimate for designing the SQL/DS tables is:

Simple table structure (One SQL/DS table per DL/I segment): (with none of the complexities discussed below) - 4 person hours per table

Intermediate table structure (Multiple SQL/DS tables per DL/I segment): (with few of the complexities discussed below) - 10 person hours per table

Complex table structure (Multiple SQL/DS tables per DL/I segment): (with many or all of the complexities discussed below) - 30 person hours per table

The above estimates assume that a knowledgeable DL/I database administrator and application program analyst are available to the SQL/DS database designer as needed.

These rules of thumb can be affected by the tools used for doing the analysis and preparing the SQL/DS data definition statements. Using tools, such as BACHMAN, can reduce the above times.

DL/I Segment to SQL/DS Table Analysis

The DL/I database inventory as described in Chapter 3, "DL/I Application Systems Inventory" on page 25 provides the necessary information for performing the DL/I segment to SQL/DS table analysis. If the tasks outlined in that chapter are completed before doing the workload analysis, a more complete and accurate set of estimates can be provided for the effort involved in designing the SQL/DS tables.

Migration issues related to specific database features are covered in detail in Chapter 5, "Database Design" on page 53. This chapter should be read before completing the database design workload.

The major and most time-consuming task is translating the DL/I hierarchical segment structure to a normalized SQL/DS relational structure. The number of physical DBDs, the number of segment types per DBD, and the number of fields per segment type in the DL/I data base are key factors in this translation. The SQL/DS database design effort, thus, can be categorized roughly by the number of segment types and number of fields in the DL/I database that is being migrated.

Another less quantifiable metric is the degree to which each DL/I segment type migrates to an SQL/DS table in a straightforward manner; i.e., how good the 'fit' is between DL/I segment types and SQL/DS tables.

Assuming the DL/I database is well designed, normalized, and no new data fields are being added at the same time as migration to SQL/DS, it may be possible and desirable to have one SQL/DS table for each DL/I segment. The pros and cons of this approach are discussed in 5.1.1, "One-For-One Approach" on page 53. In this simple case the only thing necessary to add to the SQL/DS table which is not in the physical DL/I segment is the fully concatenated key of the segment and its parents. If all of the fields are not defined in the DBD segment it is also necessary to add these to the SQL/DS table.

The following items are an indication of the viability of this approach:

- There is no redundant data in the DL/I database.
- There are no redefines in the segment.

  This may not be apparent from the segment layout but can usually be determined from the structure definition in a copy book, or ultimately in the application program itself.

  If there are redefines used in the segment it may be necessary to add columns to the SQL/DS table to support the different fields. It may also be necessary to have additional tables.
- All fields in the segment are accounted for and are used as defined by the application programs that reference the segment.

  Frequently DL/I database designers will use filler fields or slack fields in order to avoid having to make changes to existing DL/I databases to support new requirements. This practice is unnecessary in SQL/DS. However, sometimes application programs will be using using these slack or filler fields to store data. These must be accounted for in the new SQL/DS system. These fields are also the most difficult to find because they are frequently not documented anywhere except in the application program code.

Further analysis may reveal additional complexity that exceeds the standard predictions based on number of segment types. Note that extensive knowledge of the application and its operation is usually needed to resolve issues in these areas.

Primary/Foreign Key Analysis

The determination of SQL/DS primary and foreign keys is a substantial part of the database design and is influenced directly by:

- The number of physical DBDs and segments in the subsystem being migrated.
- The hierarchical structure of each DL/I database.
- The sequence fields defined within each segment.

The correct determination of the primary and foreign keys and the corresponding referential constraints provides assurance of the integrity of the SQL/DS table design. These subjects are discussed in greater detail starting at "Referential Integrity" on page 67.

Field to Column Analysis

The number of fields per segment type is an important indicator. A high number may indicate unnormalized data which could change the ratio of DL/I segment types to SQL/DS tables.

Also consider the existence of any indirect field mapping situations identified in the DL/I database inventory:

- Field redefinitions
- Program defined fields
- Inconsistent formats
- Redundant fields.

Repeating Fields

The use of OCCURS and OCCURS DEPENDING is a field mapping situation that should be emphasized because of the potential complexity added to the SQL/DS design. Each use will require close evaluation and generally will require new SQL/DS tables or columns.

Coexistence Issues

The impact that coexistence issues would have on the SQL/DS database design itself are normally minimal. Review Chapter 10, "Coexistence Strategies" on page 161 to ensure this is the case for the specific application being migrated.

4.2 Application Program Migration Sizing

The factors relevant to determining the size of the application program migration effort, both online and batch, are based on the following assumptions:

- The program inventory is complete.
- The program is still valid.
- The source code being migrated is the same level as that used in production.
- The compiler for the language in which the program is written will reproduce the production object code.
- All pending program changes and function enhancements are frozen until after production cutover.
- Redesign is limited to the complex cases described below.
- For online programs the screens are not being changed. If the screens are being changed the normal program estimating procedures used within the organization should be used. The fact that the data will be stored in SQL/DS rather than DL/I should have minimal or no effect on screen design.

  It is unlikely that the DL/I to SQL/DS migration is being made without a desire for enhancing the system being migrated. This fact could lead to the desire to change screens at the same time and, if this is the case, allowance must be made for this fact in terms of additional time and resources.

4.2.1 Application Programming Rules Of Thumb

The reader may desire to review Chapter 7, "Application Program Migration" on page 85 before sizing the workload for migrating the application programs.

It is important that some metric be chosen to estimate the migration effort. In some installations, program size has been found to be a good indicator of the migration programming effort. Smaller programs are generally easier to understand, manage and migrate. The program migration should be only a small fraction of the original development effort. Many of the modifications will actually delete application code related to segment-at-a-time processing and searching. The workload sizings, being a general estimate, will not be accurate for all migrations.

In addition to program size, a good indicator of the level of effort required to migrate the program from DL/I to SQL/DS is the number of DL/I calls, the complexity of the DL/I calls, and the number of DL/I databases referenced by the program.

Where there are a large number of programs and the estimate must be reasonably accurate, it is recommended that a few programs be selected which are representative of the size categories. Migration of this selected set will provide a metric appropriate to the specific installation.

The sizings are based on the following assumptions:

- The application programs are written in COBOL or PL/I.

  DL/I supports programs written in RPG and SQL/DS does not. Since RPG programs will have to be rewritten in another language, the following estimates are not likely to be accurate.
  Application programs written in assembler tend to be more difficult to migrate and thus the following estimates are probably not high enough.
- Test data are available.
- System support, though not included in the sizing, is available.
- No adjustment for a learning curve or orientation is made.
- Design, code and unit test are included in the sizings.
- No application enhancements are included in these estimates.

COBOL or PL/I Programs

The size of the COBOL or PL/I programs can be determined from the number of lines of code in the production source library. The number of DL/I calls can be determined from an inspection of the application code. The number of DL/I databases referenced can be determined from the PSB associated with the program.

This information will be readily available if a complete application program inventory as described in Chapter 3, "DL/I Application Systems Inventory" on page 25 has been completed. The application program information will be found in described in Table 14 on page 42. The PSB information is contained in Table 15 on page 44.

An approximate migration workload estimate for programs is:

Small: (less than 750 lines, with 1 to 5 DL/I calls referencing 1 or 2 DL/I databases) - 12 person hours

Medium: (from 750 to 1500 lines with 6 to 15 DL/I calls referencing 3 to 5 DL/I databases) - 40 person hours

Large: (more than 1500 lines with 15 or more DL/I calls referencing more than 5 DL/I databases) - 80 person hours

Complex COBOL or PL/I Programs

In DL/I, navigation (finding and selecting the database segments for the application) is done by logic in the application code, or through the use of segment search arguments (SSAs) in the DL/I call, or a combination of both. In SQL/DS, this is done with an SQL statement. The search conditions in the SQL statement identify the data that is to be retrieved.

In DL/I it is possible to retrieve multiple segments with one DL/I call by using the path call function. This may lead to the situation where one DL/I call must be replaced with several SQL statements. Whether or not this will occur is very much a function of the design of the relational tables. It should be considered carefully as it can lead to a complex relational application that was a relatively simple DL/I program.

One good way to handle a situation like this is to take a random sample of COBOL or PL/I programs for detailed analysis. Closely examine the samples looking for:

- Loop processing around DL/I calls
- Segments accessed but not processed
- Complex DL/I calls using the full power of the SSAs. (Such facilities as command codes and complex Boolean operators are indicative of complexity in the SSAs.)
- Complex iterative processing.

The presence of any one or all of these conditions in a program could indicate a complex migration. If the program is complex, it might be more cost effective to redesign for a relational solution.

By analyzing a small sample of the entire set of application programs it is possible to determine what proportion of the sample fall into each of the small, medium, or large categories previously described. It is then possible to extrapolate the time estimate for migrating the entire set of programs without letting complex COBOL or PL/I programs distort the estimating process.

CICS Online Programs

The assumption in this chapter is that screens are not being changed as part of the migration. There are other considerations, however, that must be dealt with such as:

- Conversational Programs
- Pseudoconversational Programs
- Online or Batch
- Synchronization of DL/I and SQL/DS database updates

4.3 Special Programs and Procedures

The data conversion plan will require some level of special purpose programs and procedures. Each aspect of the data conversion should be evaluated to provide direction for estimating the effort involved. In some cases a similar program may exist that can be adapted. Estimating guidelines for these programs will be similar to those described in the preceding section for application programs. New programs identified should be estimated according to installation standards for new application development.

4.3.1 Data Conversion

One of the areas often overlooked which can cause a significant impact on a migration schedule is the effect of 'dirty' data. Many users assume that their system contains only 'clean', consistent data. Migrating to a well designed relational database can be very troublesome if the data is not 'clean.'. This can significantly impact a migration schedule. Therefore, an effort should be made to verify the integrity of the the existing data. Refer to 6.2, "Loading Tables" on page 83 for further discussion of this topic.

The physical reformatting and transfer of the data from the DL/I database to the SQL/DS database will require special programs and procedures. Chapter 6, "Data Migration" on page 77 describes the requirements for these programs and some available options.

Data conversion requirements can be categorized as:

- Extract - unload data from the DL/I database
- Format - reformat the data from DL/I to SQL/DS table format
- Load - load extracted data to the SQL/DS database
- Validation - verify the results of the data conversion.

  Some techniques for verification are discussed in 6.3, "Data Conversion Validation" on page 84.

Some of these functions may be combined in a single program. Most of the effort will focus on programs to extract and format data, but procedures for DL/I, SQL/DS, and system utilities will be necessary. Special reporting programs and queries may also be required, particularly for data validation.

4.3.2 Coexistence Programs

Additional programs may be required to support coexistence of the two database managers as the migration is staged. Applications which will use data on both database systems and data which is related across application boundaries could result in additional programming effort.

The use of coexistence programs as a migration tool is a powerful tool for staging the effort. Potentially there can be an added cost that occurs when one has to migrate an application program to access both DL/I and SQL/DS databases during one phase of the migration, and then later modify the same program again when all of the data have been moved to the relational environment. Depending on the number of migration phases, this could happen several times. This is most likely to happen with highly integrated DL/I systems.

Properly planning the migration stages can minimize the duplication of effort described above.

Bridge Programs

Bridge programs allow access from an DL/I application to an SQL/DS database and/or vice versa. If the application requires updates to both databases, and this update cannot be performed within a single unit of work, there may be integrity concerns in the event of a system or program failure. This is discussed in Chapter 10, "Coexistence Strategies" on page 161.

Parent/Child Segment Relationship Splits

If there is a one-to-many relationship between an DL/I parent segment and its child segment (as is usually the case), it is likely that the data from these two segments will reside in different SQL/DS tables.

An added complexity arises when it is necessary to split parent/child segment relationships such that one remains on DL/I while the other is migrated to SQL/DS. See 5.5, "Designing for Data Related Between DL/I and SQL/DS" on page 71 for further discussion on this topic. There will be special programming and database design considerations that will depend on the usage and installation considerations. Refer to Chapter 10, "Coexistence Strategies" on page 161 for the requirements and considerations.

Synchronization Programs

In the event that some of the data are to be replicated between the DL/I system and the SQL/DS system, programs may be required to synchronize the two copies of the data. This is discussed in Chapter 10, "Coexistence Strategies" on page 161.

Chapter 5. Database Design

This chapter discusses the task of migrating hierarchical (DL/I) databases to relational (SQL/DS) databases and the approaches that can be taken. As introduced in Chapter 2, "Migration and Coexistence Considerations" on page 11, there are several paths you can follow to accomplish data and application migration. Your approach may be a modification of the alternatives described. Your individual requirements will dictate the methods you use that will accomplish your business needs. The techniques that can be used are reviewed below, specifically in the context of database design tasks, and the merits of each approach are evaluated.

5.1.1 One-For-One Approach

This approach can be referred to as a "quick and dirty approach." DL/I segments are mapped directly to SQL/DS tables on a one-for-one basis. The assumption is made that the data within the segments is already normalized and that each segment will make a good table.

Advantages: With this approach, the migration is simplified because the changes to the application programs can be minimized. Changes to the data (data type and size) are minimized. The data being moved from the hierarchical structure to the relational tables is moved with few changes. DL/I calls are replaced by SQL statements which deliver to the program workareas the data formerly obtained from DL/I databases. Minimal knowledge of the data is required for this type of migration.

Disadvantages: This approach is unlikely to provide optimum performance characteristics in the SQL/DS environment and loses the productivity advantages of the relational model. Thus, while migration itself is simplified, it is possible that the end result will be unsatisfactory for either current performance or future additions. The assumption that the DL/I databases are normalized is not necessarily valid. Many DL/I databases date back to periods when data modeling techniques were not used rigorously. Also, many segments contain fields that were never used and fillers that were to accommodate future requirements. Also, the DL/I databases are often denormalized for performance reasons. Carrying forward these design restrictions to an SQL/DS environment is not desirable.

Reasons for Using: It may be desirable to migrate all DL/I databases to SQL/DS in order to discontinue the use of DL/I, and at the same time achieve some of the advantages of relational. This may be an interim step prior to redesigning the applications and databases that are presently on DL/I. A one-for-one migration can be an initial step, but the resulting migrated databases should not be considered the final product. This is an interim approach and should be considered as such.

5.1.2 Redesign Approach

This is the most comprehensive approach and involves analysis of existing databases and application programs. The data requirements of the application are determined and a new data model is developed. From this model, relational tables are designed. Program requirements are determined and new programs are developed using traditional coding techniques, reporting tools

such as QMF, and possibly other tools such as application generators. Existing programs may often be modified to access the new relational data structure, instead of being replaced by new programs.

Advantages: The major advantage is that the data is structured to make effective use of the relational environment. The data is normalized and tables are created to support the new data structure. Problems in the hierarchical data structure may be eliminated and the new structures may take full advantage of the relational model. New programming tools may be used (such as QMF and CSP) which could decrease development time and increase the ease of program maintenance.

Disadvantages: Application programs are greatly affected by this approach. In most cases new programs must be developed. Some of the existing programs may be modifiable, but many programs will need to be redeveloped.

Reasons for Using: New business requirements may dictate the development of a new data model, and redevelopment of existing applications, and there are definite advantages of moving to a relational environment. Performance may be critical; therefore, the relational databases must be designed to provide acceptable response, throughput, and resource utilization.

5.1.3 Coexistence Approach

With this approach (which could be used in combination with other approaches), data is replicated in both environments, or some data is extracted from the DL/I databases and loaded into SQL/DS tables. Data propagation becomes an issue in order to maintain the data content of both sets of databases. A coexistence strategy is discussed in detail in Chapter 10 of this document.

Advantages: Existing applications continue to operate without change. New programs can be developed for the relational environment to provide user access to data in ways that change as business requirements change. New tools are available to the end user or to the programmer for accessing data and preparing reports.

Disadvantages: The data in both data structures needs to be maintained in parallel. Additional effort will be required to keep data synchronized, and to keep data structures logically equivalent if new elements are added. Recovery of one database cannot be made without considering the impact this will have on the synchronization of the two databases. Additional DASD space is required to hold replicated data. The support costs for both database products must be considered.

Reasons for Using: New requirements may be defined and they may be best accomplished in the SQL/DS environment. Relational may also be identified as the database system for future applications. You may not want to take the time required to migrate all data and programs to an SQL/DS environment.

5.1.4 Limited Redesign Approach

Design trade-offs can usually be made and this approach is a composite of the one-for-one and redesign approaches. The scope of the trade-off will depend on the objectives for the migration, the degree to which the DL/I data is normalized, and the performance variations that can be tolerated.

The primary objective is to optimize both performance and application design. The approach involves a limited redesign of the database and changes to those parts of the applications that are product sensitive. No attempt is made to change the application data requirements. The assumption is that the application will continue to need and use the data that is currently stored. There will be cases where the designer recognizes that better ways exist to store the data or that some other change should be made to improve the format or structure of the stored data. Despite the temptation to make changes that appear minor, the designer must recognize that making such changes can seriously impact applications. Assuming that an objective is to minimize the time required to migrate, such changes should be delayed until the application enters the maintenance cycle.

Availability of one or more persons fully familiar with the data and its usages is required because documentation rarely provides all the information necessary to do an intelligent migration.

This approach provides a framework for the remainder of this chapter.

5.2 Database Design Philosophy

At first glance it seems possible to treat each DL/I segment type as a relational table. It would appear that data items stored together in a segment are related, so it might make sense to store them together in the same table. In fact, it quickly becomes apparent that this is not always possible. Conditions that are not acceptable in a relational table may be acceptable in a DL/I segment. While a DL/I segment may be normalized, in many cases it has not. To achieve normalization, in some cases, it is necessary to split segments into multiple tables or, conversely, combine multiple segments into a single table. There will even be some cases where the segment may be discarded without migration.

A relational system allows a user to interrogate any column by name, based on its data value. Data formats (such as COBOL REDEFINES) that are acceptable in a DL/I design are not supported in a relational table.

The DL/I segments become a starting point in a relational design. The intent is to make a table from each segment type but each segment will be analyzed as to normal form, redundancy, and conformance with relational standards.

5.2.1 Design Information Sources

The DB/DC Data Dictionary is a primary source of information, if it is still used and maintained. If the DB/DC Dictionary is not used, an analysis of the DBD's and segment description layouts (copybooks) must be made. DL/I does not require all fields of a segment to be defined in the DBD. In fact, it is recommended that (in addition to sequence fields) only the fields referenced in segment search arguments (SSA's) be defined. Also the fields that are defined may overlap or be broken into subfields. The segment description layouts

(from copybooks) should have precedence over the DBD's because they fully describe the content of the segments. However, the DBD will identify the presence of a sequence field and whether or not it is unique. Field names from the copybooks could possibly be used as column names. Picture clauses can be used to establish data types and column sizes.

In DL/I, the unit of data manipulation is the segment, with processing taking place on the entire segment. Internally, DL/I performs field comparisons on a byte-to-byte basis. No check is made by DL/I to ensure that the data contained within a field is of the type specified by the FIELD operand of the SEGM statement. The exception to this rule is when the defined field is used with field sensitivity. By contrast, in SQL/DS, the amount of data to be manipulated can be restricted to a single column in a single row. Data comparisons distinguish between numeric and non-numeric data types, with any attempt to store non-numeric data in a numeric column being rejected.

If reproduction of every aspect of DL/I is attempted in SQL/DS, all data defined to DL/I as character data should be migrated as character. This, however, would limit the capabilities and usefulness of SQL. For example, built-in functions that operate on specific data types could not be used; e.g.SUM and AVG cannot be used with character columns.

A comparison of DL/I and SQL/DS reveals the following additional differences:

- If the application puts invalid data in a field, DL/I ignores it. SQL/DS reports it and does not perform the update.
- If the application puts data in a filler field, DL/I knows nothing about it, but will store the data if it's in an I/O area referenced by an insert call (ISRT). The concept of a filler field has little or no value in SQL/DS. SQL/DS only stores data that is from an explicitly defined host variable that is referenced by a VALUES clause, or from a literal within the SQL INSERT statement.
- If the application contains nonstandard definitions of a data area, DL/I knows nothing about it. Depending upon the data types, SQL/DS either reports it and does not perform the update, or splits data belonging to one field among the columns which were defined in the course of data migration.

5.2.2 Design Sequence

The following are the steps required to migrate DL/I data to SQL/DS data:

1. Verify that each DL/I segment is in third normal form. If the segment is not in third normal form, then the entire database record (all segment types) should be reviewed before making any attempts to normalize one segment. It may be that the entire database record has to be normalized.
2. For each DL/I segment that is already in third normal form, create an SQL/DS table.
3. Translate each DL/I field to an SQL/DS column. There are fields that may be composite fields and they should be split into several columns. Also, some fields such as part numbers have meaningful codes within the part number. Additional columns could be defined in order to separate these codes. In this situation, the part number would stay intact and the columns with the codes would be duplicate data in new columns.

4. Identify a primary key for all tables which represent former DL/I parent segments. Not all parent segments within a database record will have sequence fields. Some thought will have to be given to developing a unique primary key for the tables developed from these segments.
5. Identify primary and foreign keys for each table that represents former DL/I dependent segments. The selection of keys whether primary or foreign is similar to the discussion in the prior item.

While the above may be adequate to accomplish the design in some installations, many users will have exceptions and specific conditions that must be addressed.

5.2.3 Data Naming Considerations

There is a difference in data naming capabilities between DL/I and SQL/DS. DL/I permits an 8-character field name in the DBD while SQL/DS permits an 18-byte column name in the table definition. Typically, the DL/I user does not rely on the DBD field names within the program. In fact, not all of the fields are identified in the DBD, and the fields are really referred to by the names defined for the I/O areas. These names are limited by the programming language used. In some cases the names used for DL/I fields could be longer than the 18-byte column name limit for SQL/DS.

Care should be taken when reducing the length of the field names because duplicates could result from an approach that truncates the DL/I field name. The user should determine the best approach for their installation. It may be necessary to rename all of the fields. In this case the discussion on naming standards should be reviewed for SQL/DS columns. Regardless of the approach that is decided upon, field names will have to be changed before the application program migration is begun. Name changes should be coordinated between the application program and the database migration personnel to ensure that all parties are aware of the new field names and standards.

The following table shows some of the major terms and definitions and associated naming conventions for SQL/DS. Ensure that the names you assign when migrating your databases follow the rules as given in the table.

Term	Length (Bytes)	Description
Database	8 maximum	Name of a database that contains one or more DBSPACEs, its related tables and indexes and the associated catalog tables, log and directory.
DBSPACE	18 maximum	Name of a logical space allocation that contains one or more tables and their indexes. The name must be unique within a creator id.
Table	18 maximum	The name of a table. It is used only within the relational database system and is not known by the operating system. It must be unique within a creator id.
Index	18 maximum	The name of an index. An index is always directly related to a table.
Column	18 maximum	The name of a column within a table. It must be unique within a table.
View	18 maximum	The name of a view for one or more table. It is used in the application programs.
Package	8 maximum	The name which designates a package.

Table 16. Relational Database Terms and Name Structures.

Some additional suggestions applicable to DL/I to SQL/DS migrations are as follows:

- Incorporate into the trailing portion of names of indexes a character indicating its role: XnP for a primary key index (which normally will also correspond to a HISAM index or HIDAM index DBD); XnF for a foreign key index; XnS for an index serving the same purpose as (or corresponding to) a DL/I secondary index.
- Incorporate a (possibly shortened) version of a DB PCB name into a view name, if the purpose of the view is to correspond to a DL/I PCB.

5.3 Hierarchical-to-Relational Considerations

It is useful to understand a few of the similarities and differences between hierarchical and relational database models. While the relational model can perform all of the functions of a hierarchical model, the two are quite different in concept. In particular, the relational model is much more rigorous in its definition and conformance requirements. The hierarchical model is much more structured and uses the structure to convey information.

Relationships: Relationships in a hierarchical structure are based on definitions to the system. These relationships are implemented by an imbedded pointer structure consisting of addresses of the related segments. This fixed type of structure provides direct paths along relationships but does not support relationships that change as the data values change.

A relational database permits the free association of data based upon any data values present in the tables. Normally such associations are based upon keys. The relational model defines two specific types of keys: a primary key, which identifies an individual row, and a foreign key, which provides a reference to a primary key in a row in another table. These two types of keys are used to provide the kinds of relationships supported by hierarchical links. However, these keys may not be physically stored in an imbedded pointer structure, and they must be added to the table definition during the logical design process.

Referential integrity: DL/I databases maintain data consistency through rules implemented in the hierarchical definition. The relational implementation of referential integrity controls data consistency based on data values (primary and foreign keys) in related tables. As linkages are replaced by keys, data consistency rules may be specified which serve the same purpose as the DL/I rules.

Ordering: DL/I databases permit the concept of physical segment ordering within the segment types of the database. This ordering is foreign to the relational concept. Where physical ordering is assumed in the hierarchical model, there must be a substitute capability provided in the relational implementation. Ordering defined on some data element in the segment (i.e., where a sequence field is defined) can be readily implemented, but the problem may become complex in cases where the physical order (which is the assumed retrieval order) is chronological or programmer-controlled, rather than based on data values.

DL/I multiple-occurring segments that do not have a sequence field, will need a primary key in SQL/DS. A counter value or a sequence number may need to be used to create a primary key. Using a counter is helpful in cases where the creation time is not important, for example, when creating text. The highest number of the counter can be retrieved by SQL functions. The use of an increment value greater than one is also recommended, to facilitate future maintenance of data (such as insertion of a row that is logically between two existing rows).

Where creation time is relevant, the use of TIMESTAMP is recommended. In this way you may use date and time for ordering the data in the same sequence as the ordering of segments within DL/I.

Segment/table format: DL/I database segment formats are less restricted than relational table formats. A relational table must preserve the capability of a user to search on any column value. This restricts the format of the table to ensure that the search column can be identified by the database manager without ambiguity. Thus, the REDEFINES clause that is used in some hierarchical structures, is prohibited in a relational table. For the same reason, and also to enforce normalization to the degree possible, repeating fields and groups are prohibited. Where the DL/I structure violates the rules of SQL/DS, a redesign or circumvention must be developed.

Navigation and position: These are special concepts in DL/I. They provide "structural integrity" and give rise to three major differences between the hierarchical and relational data models.

First, DL/I may use pointers to move from one occurrence of a segment to another. The use of pointers in DL/I is implicit; in SQL/DS, the system does the navigation, based on search criteria and decisions by the optimizer. The optimizer may elect to use an index or to ignore it. The user asks for data based on tables, column, and search criteria not on how it is stored. A user may not directly specify how navigation is to be done.

Second, the order of presentation of segments to a DL/I application program which issues a series of unqualified get next calls follows the hierarchical sequence through the database record. Before the next occurrence of a segment type is presented, all of its dependents are presented sequentially. If any of these dependents have dependents, then those dependents are presented prior to the next occurrence of the the first dependent segment type. This type of sequential retrieval is not inherent in the relational model.

Third, there are some special rules governing the positioning of DL/I after execution of a database request. DL/I's position after execution of a database request is usually on or immediately after the last segment to be manipulated. But if a database request was unsuccessful, the position in DL/I can be different. For instance, if a database unqualified get request was unsuccessful, the position after the call will vary depending upon factors such as the actual contents of the database and the nature of the request which was unsuccessful.

Application dependencies on such navigation and positioning characteristics must be carefully examined and then modified or removed, if the data is migrated to SQL/DS. The concept of positioning does not apply in SQL/DS. Positioning may be simulated by the use of CURSOR, but probably not to the same degree as positioning in a hierarchical database.

5.4 Logical Database Design

This section addresses the problem of generating a design which maps all of the data and capabilities of the DL/I database into a set of SQL/DS tables. The intent is to identify all known types of problems and to describe possible techniques for their solution. In many cases there will not be a single best solution. It will depend upon the use of the data by the application.

The designer will find differences in the ease of redesigning different databases due to a number of factors. An independent factor will be the "age" of the database. The older the design of a database, the greater the probability that design compromises have been made with the addition of new application systems. These sorts of conditions will tend to complicate the redesign of the database. At the same time, these are the databases that will gain the greatest benefit from a redesign.

5.4.1 Designing Tables

The first step in converting your databases is to develop a logical design for the structure of the new database system that you want to use. There are general considerations that can be applied to each of your databases. However, each database must be treated separately.

Logical Design Steps: The following steps consist of general recommendations and procedures for producing a logical design:

1. Define which databases and what data should be available in the relational database system. Your existing DL/I database definitions can be used as a starting point for your relational database design.
2. Develop an enterprise-wide conceptual data model before you do your logical database design. Consider the future requirements of your organization.
3. Analyze the existing DL/I databases. It is especially important to find out how the application programs' functions are using the data. For example, what segments are accessed and for what purpose (i.e. read only, update or delete).
4. Consider during database design the impact of your design on existing application programs. The data type DATE in the relational system will require application program changes. Dates are handled differently in the relational database system. In SQL/DS, you use only the data type DATE in tables when defining date data. All other methods of storing dates can be eliminated. This also applies to timestamp operations. Internally to the application programs you still could use your old date format to avoid migration efforts in the programs. Built-in functions are provided to convert the data retrieved from the tables to other formats before processing.
5. Document the new databases and all data changes. Include such information as the design decisions, the reasons for the decisions, and the impact of the design on data migration and program migration.
6. Design your relational tables and table relations.

5.4.2 Designing for Relational

The migration team must have available a database designer with skills in SQL/DS performance issues. Performance issues must be addressed with respect to the specific applications being migrated.

Undefined Fields

Some DL/I users will have documented all of their data in a data dictionary, but there are some users who do not have a data dictionary or who have not maintained their dictionary. There are other sources for the necessary information. DBD's typically only document the fields required by the database manager. The copybooks for each of the segments should identify all of the fields in the segments, and these copybooks ultimately will be the best source for documentation. It is necessary to have a team member or have access to someone who can verify that the defined fields are used by the applications. It is possible some defined fields are not used by any of the application programs. This analysis should be done while the applications are being inventoried. The inventory will then contain the fields to be included in the SQL/DS table definitions.

Normalization

It is not our purpose here to discuss techniques to normalize the data nor the value of doing so. *An Introduction to Database Systems* by C. J. Date contains an excellent discussion of all of the normal forms and their value in data processing. Suffice it to say that normalization attempts to isolate attributes with their unique primary key. The result is a set of non-redundant tables with high consistency and flexibility.

Older databases may have a number of design compromises that have been made to accommodate the addition of new applications or for performance reasons. For instance, common data may appear in multiple segments. Such compromises, while appropriate for DL/I databases, may be counter-productive for SQL/DS databases. Normalization will uncover the compromises that need to be examined. SQL/DS table design may also require some compromises of normalization for performance reasons, but the nature of the compromises will undoubtedly be different from those implemented in DL/I.

When a segment is retrieved in DL/I, the value of its sequence field is placed in the key feedback area. When a dependent segment is retrieved, the value placed in the key feedback area is the concatenation of the sequence fields in the path to the segment. This concatenated key serves to identify a segment uniquely when all segments in the path have unique sequence fields.

Since DL/I works with pointers, it does not need to store the sequence field values of parent segments with the dependent segments. Relational systems work only with data values, so any such relationship must be defined explicitly. Consequently, rows in the table for the dependent segments need to contain columns for the sequence fields of the parents in addition to the sequence field of the segment itself. These then become foreign keys (as well as part of the dependent row's primary key, in many cases), allowing data in *dependent tables* to be related to data in a *parent table*.

The relational model is based upon normalization. Therefore, to achieve its benefits it is best to attempt to normalize the data. A simple conversion may be done without normalization, but the result will not be as effective as when normalization is done. The limitations of unnormalized data will result in reduced flexibility, especially for ad-hoc end users.

Repeating Fields

Repeating fields of a segment (discovered, for example in copybook definitions) may be handled in multiple ways when defining the tables for the segments. The most simplistic way is to normalize the data and generate multiple rows, one for each occurrence of the repeating group. This introduces redundancy of the columns (formerly fields) *not* in the repeating group. In most cases, a better solution is to define a new table for those attributes which can occur multiple times. This may complicate migration of application programs and could increase the volume of data. Where the data is a variable number of repeating fields, separate tables may be the only possible solution.

Where the data is a fixed number of repeating fields (e.g., quarterly sales for the last twelve months) then the fields could be translated to columns with unique names such as QTR1, QTR2,... and the program logic adjusted accordingly. Similar logic may be used even with "OCCURS DEPENDING ON" type fields where the maximum number (and all applications have some maximum even if it is the maximum allowable segment size) of fields is

translated to columns with unique names and all unused columns set to NULL. Note that there may be a considerable cost in storage for this option. Such a method loses the capability of providing indexing across the columns and will complicate searches by requiring WHERE clauses such as "QTR1 > 100 OR QTR2 > 100 OR ...". This method is not particularly beneficial to end users; the *position* of the data is conveying information, so the user must know the meaning of (e.g.) QTR1 versus QTR2. It is potentially even more of a problem for database administration. Should users decide to keep additional data (quarterly, in this example), a field (column) must be added, which could result in a reorganization. If the repeating data were in a separate table, the same result would be achieved by inserting a row - usually avoiding any requirement for the DBA to be involved.

One method that will conserve storage and minimize program impact is to take a field originally defined to DL/I as something like:

```
05 MONTH-ACTIVITY OCCURS 0 TO 12 TIMES . . .
    PIC X(04)
```

and define it to SQL/DS as:

```
MONTH_ACTIVITY VARCHAR (48)
```

By modifying the program so that it begins with an initial move from the input area to another structure with the original definition, the program logic is essentially preserved. This technique naturally loses all the relational capabilities for data in this column and obscures the true nature of the contents. The technique should be avoided wherever possible.

Overlapping Fields (REDEFINES)

Inspection of the copybook definitions of segments may uncover overlapping (or COBOL "REDEFINES") types of fields. Typically this condition is the result of "subtypes" of segments. It is common to find multiple types of an entity. For instance, an application may address multiple types of products that a company sells; manufactured at this location, manufactured elsewhere (plant name and shipping information required), and purchased (vendor and price required). A single database record may well redefine the content for the three subtypes. In cases of redefined segment content, it will be necessary to analyze the data and determine whether multiple tables should be defined or whether all types of fields properly belong in the same table. This will require normalization of the data to determine the correct association. It will also require the assistance of someone familiar with the application and its associated data. The first test that can be applied is to determine if part of the primary key is redefined. If so, the data is not normalized and should definitely be divided into multiple tables with unique primary keys. In some cases, data may be unnormalized but still have a single primary key for the attributes.

If the record is in normal form (specifically third normal), then the redefined data must be mutually exclusive (fourth normal form would generate multiple table types for this condition). This is the case with "subtype" information as described above. It may or may not be appropriate to go to fourth normal form, but the design of the DL/I segment implies that third normal form will be more effective. In an SQL/DS database, each field must be defined as a separate column in the table. The program will have to be modified to cause one to be "null" while the other has a value. Which fields should be expected by the

program will normally be determined by some code field in the segment which the program interrogates.

If the data reflects unnormalized data, it will probably be desirable to create a different table. It will be necessary to understand how the programmer determines which of the fields exist in the DL/I database. Decisions on how the programs are to be modified to address two tables instead of the single segment type will determine whether an identifier is carried into the new table.

In any case, a REDEFINES condition will require joint planning between the database migration and the program migration groups to ensure consistency in the solution.

Splitting One Segment into Several Tables

There may be situations when you have to split one segment into more than one table. The reasons for this might be:

- Part of the data should be a multiple occurrence.
- Security purposes requires this action.
- Some of the data does not always exist.
- Data is placed in separate tables to provide optimum access by ISQL or QMF users.

The primary key of both sets of data is the same (or, in the case of repeating data, the key of one table is a superset of the key of the related (parent) table), and a foreign key connection usually exists.

Variable Length Segments

Variable length segments can migrate to a relational table with relative ease. If variable and fixed length fields are mixed in the same segment, working with variable length segments is more difficult, but not impossible. In this case, the variable length fields should be defined as the last columns in the table. In most cases the variable length segment is comprised of one field such as an address, and this will migrate to a relational table as a column of data type VARCHAR (variable-length character).

Secondary Indexes

In DL/I, a secondary index is essentially a database containing pointer segments that are used to provide an alternate entry into the database record. In some cases, the hierarchical structure the programmer sees is altered if the target segment is a segment other than the root segment. Also, the secondary index can contain data that is replicated from the source segment and can be processed as a database in its own right. Secondary indexes that change the hierarchical structure will require the corresponding change in the SQL call pattern to the tables that represent the segments of the database record. You should, however, be able to deal with this without a significant impact on the application programs.

With DL/I secondary indexes, it is desirable to generate a unique sequence field in the pointer segment. In cases where the sequence field is not unique in the source segment, a system-related field can be appended to the sequence field. (This system-related field can be the concatenated key of the source segment, or the RBA of the source segment.) However, the purpose of making the secondary index sequence field unique pertains to improving the handling of

indexes of and by the underlying access method (e.g. VSAM), and does not usually affect application logic. The application program could retrieve multiple occurrences for any specific source segment sequence value. The analogous processing logic would be implemented in SQL/DS, namely an OPEN-CLOSE-FETCH loop. The WHERE clause in the associated cursor would be qualified on a value for the column corresponding to the original source field. Note that if an index *were* created for this column (often beneficial for performance), it would not be unique, unlike the original secondary index. This is a case where the application structure would not change, but the data (index) design would be different: unique in the hierarchical system and non-unique in relational.

In the situation where the source field comes from a segment other than the target segment, the nature of the relationship between target and source segment would have to be examined. If it is a one-to-many relationship, a primary key-foreign key relationship should suffice to identify a parent table (target) row associated with a particular foreign key value. If it is a many-to-many relationship, then the relational design should include an association table to relate the tables for the two related entities. SQL/DS indexes again are not actually required in either case (except for defined primary keys), but could improve performance of SQL statements qualified on indexed columns.

In some cases, a 'sparse' secondary index is specified. If so, when migrating to relational you should consider how to handle exclusion of rows in a manner logically equivalent to exlusion of segments in DL/I (in addition to ordering considerations as described above). If NULLVAL was used in the DL/I secondary index, one could change the specified value to NULL in the SQL/DS design (probably best for the long term) or continue to use the value originally specified In either case, one could then use views (or SELECTs) with a WHERE clause that excludes the unwanted rows. If EXTRTN was used, you would have to examine the exit routine's logic to determine how to specify a predicate to exclude the unwanted data.

Secondary indexes allow different entries into a hierarchical structure. In SQL/DS, indexes can be defined on any column, but the final determination if they will be used is made by the optimizer. The best approach for dealing with secondary indexes is to not attempt to duplicate them in SQL/DS, but to understand why they exist in DL/I and to determine whether or not they are used. They may exist to extract data from the hierarchical database for fast retrieval. In this case another table may be called for or maybe the SQL/DS tables will provide for the application needs. They may exist solely to provide an alternative sequencing of the data; if so, this can be done with the SQL ORDER BY clause. An SQL/DS index may be desirable to improve performance of the SELECT with ORDER BY, but is not required. The key is to understand the database and its secondary indexes and then determine what is needed in the SQL/DS environment.

Redundant Data

For performance reasons, the same data may have been defined in multiple segment types. These cases should be readily identifiable. If a data dictionary is being used and maintained, you can find all cases where the same element name occurs in multiple segments. However, this will not always mean redundant data. Often a common definition, such as date, will be used in many segments with different meanings. Cases of elements that occur in two or

three segment types should be investigated to determine whether they are in fact redundant.

Physically paired logical child segments will contain redundant data. This is done to reduce the need to retrieve the data from the logically related database. The performance impact will have to be evaluated to determine if there is value in the relational implementation of duplicate data.

Where an element is found to be one of multiple copies of the same data, it is necessary to make a determination as to whether the data should be normalized to a single table or left in independent tables. Two factors will apply:

- Application impact

  If the application will require considerable program change to accommodate a single copy form of the data, it may not be economical to eliminate the redundancy. Trade-off between the cost of maintaining the multiple copies and the cost of application change must be identified. The risk of inconsistency of the data values must also be taken into account. Consider the use of synonyms and/or views to reduce impact on applications.

- Performance impact

  In some cases the same performance advantages of separating the data will apply to the SQL/DS implementation. This will have to be balanced against the cost of multiple copies of the data such as duplicate maintenance. In those cases where the data is typically very stable (e.g., customer name), duplicate copies of the data are probably acceptable. This is a typical consideration in assessing the desirable degree of normalization.

Segment Combination

Normalization may reveal that multiple segment types can be defined as a single table in relational format. You may have a situation where a single root segment has a single dependent segment. In SQL/DS, these segments can be combined into a single table. The data will now be created and deleted at the same time. In this case, the application programs will be impacted. In another situation, the root segment may contain only a key and there are multiple occurrences of one dependent segment type. To combine these segments, the key of the root segment is added to the key of the dependent segment and each occurrence of the dependent segment becomes a row in the table.

Merging segments into a single table will simplify the data structure and the applications. The creation of views on the tables will protect the programs from having to deal with irrelevant data.

A problem will undoubtedly arise in combining segments due to inconsistency of the data. This can take two forms:

- Unmatched segments

  There will be cases where not all of the segment occurrences required to make up the combined segment exist. This may be due to application error or due to the nature of the data. It will be necessary to make some decisions as to how the condition is to be handled. It may be adequate to simply create null columns for the missing data. In some cases, essential

parts of the key may be missing and it will be necessary to do further analysis of the data.

- Inconsistent data

  Errors in input data or application logic can result in data in two logically associated segments being inconsistent such as CURRENT-PURCHASES in one segment exceeding YTD-PURCHASES in another segment. A method of resolution will have to be determined for such cases. It may be possible to ignore the problem (at year end, the YTD-PURCHASES will be reset) or it may be necessary to reconstruct correct values where consistency is important to the integrity of the application. In many cases there will already be methods in place to discover and correct such inconsistencies as they come to light in various reports. In these cases, the inconsistencies may be ignored as the existing techniques will reveal and correct them.

Referential Integrity

The interdependence of data in tables and validity of that data is a primary consideration. In a set of tables, the condition in which all references from one table to another are valid is termed the referential integrity. Referential integrity, and the constraints by which it is enforced, can result in very complex effects on the tables and databases that are designed, especially when rows or columns are updated or deleted. Therefore, it is recommended that referential integrity be planned for beforehand.

Here are some recommendations and considerations that should be made when implementing referential integrity:

- Try to keep the number of constraints as low as possible. This allows more flexibility in dealing with the data. In other words, only define a referential constraint when necessary.
- It is recommended that indexes be created on foreign keys. This improves performance when checking constraints as it reduces the amount of scanning of dependent tables.
- Ensure that objects with referential constraints are defined in the correct order. (A foreign key cannot be defined unless the corresponding primary key already exists and has a primary index defined).

For full details on referential integrity, refer to the *SQL/DS Database Administration* (VM: GH09-8083, VSE: GH09-8095) manual.

The referential constraints that can be used to preserve the hierarchical integrity is ON DELETE CASCADE. This would preserve the referential constraints implied between a parent and dependent segment if the parent segment maps to one table and the dependent segment maps to a second table. If a row is deleted from the table for a parent segment, the cascade delete causes the related rows to be deleted from the table for the dependent segment. SQL/DS allows only one level of CASCADE, the dependent table that is the object of a cascading delete must not itself be a parent. Consequently if, in DL/I, a single delete call is used to erase a whole database record by deleting the root this will have to be replaced by multiple SQL delete statements if the hierarchy is more than two levels deep. The foreign key to primary key relationship prevents a row from being inserted into the table for a dependent segment unless a corresponding row already exists in the table for the parent segment.

If logical relationships were used in DL/I, there are cases where other SQL/DS options for the ON DELETE clause (i.e. RESTRICT or SET NULL) may be appropriate. One must not only consider the DL/I logical relationship delete rule specified, but also the processing options and the intent of applications that change the data using a logical DBD. In some cases, no exact equivalence exists between the implicit actions of DL/I in response to a DLET call, and the actions possible in SQL/DS in response to an SQL DELETE.

The following table provides guidance for some commonly encountered situations in DL/I, and the SQL/DS ON DELETE specification that most closely corresponds.

Table 17 (Page 1 of 2). Delete rule correlation

DL/I Delete Rule	**SQL/DS Delete Rule**
PROCOPT=delete for all segments concerned • SEGM RULES=(_P_,_) and DLET attempted on Logical Parent • Logical parent deleted if not restricted by child segment -OR- • SEGM RULES=(_B_,_) and DLET attempted via Physical Parent • Physical parent is deleted along physical path • No children restricting deletion -OR- • SEGM RULES=(_P_,_) and DLET attempted via Logical Child • Logical delete first, then physical delete • No children restricting deletion	Delete Restrict • Row deleted if no other row depends on it • Row not deleted if dependent row exists
PROCOPT=delete for all segments concerned • SEGM RULES=(_L_,_) for segment being deleted • Specific path for deletion • Segment available via another path	Delete Set Null • Nullable column of foreign key in dependent row set to null • Only immediate descendent dependent row is affected • Indicates "orphan child" rows for the relationship

Table 17 (Page 2 of 2). Delete rule correlation	
DL/I Delete Rule	**SQL/DS Delete Rule**
PROCOPT=delete for all segments concerned • SEGM RULES=(_L_,_) and DLET attempted on Logical Parent • All logical children are logically deleted -OR- • SEGM RULES=(_V_,_) and DLET attempted on Logical Parent • Done via physical path • All logical children are logically deleted -OR- • SEGM RULES=(_L_,_) and DLET attempted on Logical Child • Done via physical or logical path • All logical children are logically deleted	Delete Cascade • Designated row is deleted • Delete rule of dependent is honored if it has no dependent

Primary Key Determination

The relational model requires the existence of a primary key to ensure entity integrity, that is, the ability to uniquely identify any row. SQL/DS, however, does not require primary keys except where referential constraints are defined. In general, any segment that is a parent must be converted to a table with a primary key. Segment types which are not owners, however, need not have primary keys if there is no suitable candidate. An application requirement for ordered retrieval may, however, force a unique ordering value. Such a unique value may or may not be identified as a primary key.

The primary key must be determined by someone familiar with the data and its uses. This determination may often be made from information in the copybook, database descriptor (DBD) or data dictionary.

Once the primary key has been determined, documentation should be updated to show which columns participate in the primary key and their order. The definition of the primary key will also often result in the definition of additional columns for the table. These must be documented fully in order for application logic to reflect their existence.

Foreign Key Determination

The implementation of the relationships which were maintained through linkages in the hierarchical database is done through foreign key/primary key pairs in SQL/DS. Since the foreign key does not exist in the hierarchical model, it must be derived to implement the relational model.

Foreign keys can be identified in multiple ways:

- Parent segment unique primary key

  The primary key of the parent segment, if unique, will become a foreign key for the dependent segments.

- Parent segment non-unique primary key

  A non-unique primary key will need to be made unique by combining it with another column such as a timestamp or by appending a sequence number to the key value.

- Parent Segment no primary key

  A selection must be made of a column with unique values. In most cases a new column will need to be defined and this column can contain a timestamp or some other value that ensures uniqueness. If a timestamp is used, rows in the table may not insert initially because timestamps can result in duplicate values. The program logic would need to allow for a retry of the insert after the timestamp value has been updated.

Segment Insert Rules

In DL/I insert rules apply for segments that have no sequence field or a non-unique sequence field. When segments are inserted into a physical twin chain, the insert rules of FIRST, LAST, HERE apply. If the order of the rows in a table defined for a segment type is important, then an appropriate column (or combination of columns) must be chosen (or added) and used in conjunction with an ORDER BY clause in the SQL statements. A clustering index may be defined on this column to maintain the desired sequence even at the physical level.

Group Level Items

DL/I supports the concept of fields which are made up of a number of subfields. Date is a classic case with the subfields of month, day and year. SQL/DS does not support this concept.

There are two fundamental ways to provide this support. The most common will be to define the subfields as columns to SQL/DS. This approach is the most general and permits searching each of the subfields independently and allows the use of the subfields in different or overlapping foreign keys. Where it is necessary to address the group level item as well as the subfields, after the FETCH of the data, the individual columns may be moved with a COBOL move to a structure where they can be treated as an entity.

Alternatively, it is possible to define the group level field as a column to SQL/DS. After FETCHing the data from the table, a COBOL move can be performed to a structured work area where the data can be treated as subfields. This will, however, prohibit the use of the subfields directly by SQL/DS.

A large percentage of group level items are often found to be dates. In these cases, use the SQL/DS DATE data type. It will have some impact on programming, but any solution will impact the programming and the use of DATE will improve future productivity.

Again, once a technique is selected, it is necessary to reflect the design decision in migration documentation.

Data Security

Data security definition begins with an investigation of DL/I security and PSB's. PSB's can be translated to views and authorizations assigned. Where field level sensitivity has been implemented, the security of the PSB's can be reflected easily in views. SQL/DS security is much more comprehensive and while it may be tempting to fine tune the security specifications at this point, it is recommended that this not be done during the migration. A complete security redesign should be done with a great deal of care as it is time consuming. As such, it would slow down the migration effort.

Where very little has been done with respect to security in DL/I, it may well be adequate to grant execution on all packages to PUBLIC and use CICS security for online applications and normal machine room security for batch. Additional security will be necessary as ad-hoc users come online. For more information on this topic, see Chapter 9, "DL/I Security Migration" on page 153.

5.5 Designing for Data Related Between DL/I and SQL/DS

While every attempt should be made to avoid a migration where only some of the segments of a database record are to be converted to SQL/DS, there will be cases where it is unavoidable. This will considerably complicate both the database design effort and the application programming effort.

If it is necessary to break apart a database record in converting data from DL/I to SQL/DS, there are two basic conditions that must be addressed:

- The converted data is a dependent segment
- The converted data is a parent segment

In some cases the segment will be both a dependent and a parent. Both considerations will apply.

5.5.1 Migration of Dependent Segments

If the converted segment is a dependent segment, the relationship to the parent must be maintained and integrity rules enforced between the SQL/DS tables that represent the former SQL/DS segments. This will require that the dependent segment table contains adequate information to address the specific parent segment occurrence. Navigation will require parent information whenever a program requires access to the SQL/DS parent from the former dependent. Integrity rules will require parent information to verify the validity of inserts. This is the equivalent of a relational foreign key and may be implemented in the same manner providing the key contains all of the information necessary to navigate to the parent. In some, but not all cases, this will be a subset of the primary key of the table.

A more complex consideration is those programs which cause ownership changes or deletions (explicit or implicit) and are not converted. This group of programs must be changed or new programs supplied. If these programs remain in DL/I with the same functions, it may be necessary to change the DL/I database to incorporate a key for access to the converted data. In many cases, this key may be derived from the keys and ordering fields of the DL/I segments on the path.

An additional consideration may be whether or not it is appropriate to replicate some of the data in both systems. For performance, system isolation or integrity reasons, it may be appropriate to store some or all of the data under both systems. Data consistency will then be the issue. More information on this subject is in Chapter 10, "Coexistence Strategies" on page 161.

5.5.2 Migration of Root Segments Only

Where only the root segment is converted and the dependents are not, it will probably be advisable to leave a dummy segment as the root segment. In this case, it will be necessary for the SQL/DS table to contain key information that permits access to the dummy. The dummy segment may contain not just a key, but also static information to reduce cross system access for some programs. All inserts and deletes done in the SQL/DS table must have corollary inserts and deletes of dummy root segments. Error conditions must be taken into account.

5.6 Physical Database Design

After finishing the logical design, the next step is to do a physical design. The physical design of your databases involves the actual definition of the DBSPACEs, tables and indexes themselves.

For more information about implementing your design, including, for example, acquiring DBSPACEs, creating tables and indexes, refer to *SQL/DS Database Administration* (VM: GH09-8083, VSE: GH09-8095).

5.6.1 Physical Table Definition Considerations

5.6.2 Table definition

When performing the physical design of the relational tables you should do the following:

1. Determine the unique primary keys.
2. Check whether or not a column only depends on the whole key.

   If not, you should change your tables and try to achieve the third normal form. You can only leave it as it is if any special application programming reasons exist.
3. Check field interdependence and allowed data.
4. Analyze the secondary indexes and decide:
   - Is a separate table required.
   - Are equivalents for a suppress routine or a null value required.
   - Is user data stored in the secondary index and, if so, how is it used.
5. Multiple dependent segments need a key for the root and dependent segment.
6. Single dependent segments. Allocate segments occurring only once:
   - To the parent (unless its occurrence or application does not support this).
   - Otherwise add a root key.

Check whether or not there is a potential for multiple occurrence in the future.

- Note that dropped segment types will have an impact on the programs that delete and insert this data.

7. Root segments without data are not needed.
8. Derived fields, e.g. fields which contain totals of other fields, usually can be dropped.
9. Each column should be checked for:
   - Field name and length.
   - Field attribute.
   - Defaults.
   - Null value.
   - Index fields.
   - Cluster index fields.
   - Foreign key and/or primary key.
   - Sort sequence (ascending, descending).
   - Referential integrity and entity integrity rules.
10. Check whether and where the database can be made simpler:
    - Remove repeating groups.
    - Remove redundant and single occurring data.
    - Merge data of various databases.
    - Use the referential integrity functions.
    - Remove indexes and pointers.
    - Remove multiple segment and field layouts.
11. Check which programs can be dropped or replaced.
    - Some of the batch programs to initialize the databases can be dropped. Initialization is not necessary for the relational tables.
    - Load programs can be replaced by utilities in the relational database system.
    - Some programs can be replaced by QMF queries, forms, and procedures.

5.6.3 Storage Pools, Databases, and DBSPACEs

Use of SQL/DS physical constructs such as storage pools, DBEXTENTs, and DBSPACEs are not a migration issue. Nothing in the DL/I design will constrain or influence the choices made for these objects. All normal SQL/DS design considerations apply.

5.6.4 Indexes

Root segments in a HIDAM database must have a unique key field. An index entry for each root segment is created in a separate index database. An index should be created on the same primary key in the SQL/DS table as the one that was defined for the root segment.

HDAM does not require a unique key field. An index could be defined on the key field, but the use of an index should be dictated by performance considerations. In the situation where the the key field of the HDAM database is unique, an index should be defined on the primary key of the table corresponding to the root segment.

In situations where the insert rules of FIRST, LAST, and HERE may determine how segments are retrieved, appropriate column(s) must be chosen or added and used in an ORDER BY clause to duplicate the DL/I retrieval sequence. An index on columns in the table corresponding to these segments may need to be created to improve the performance of this retrieval. The optimizer determines whether the index will be used[3] for a specific SQL statement.

5.6.5 Concurrency

The benefits of an installation standard such as requiring the use of CICS pseudo-conversational transactions may be evaluated here. With pseudo-conversational, CICS will issue a commit (syncpoint), thus release page locks before sending the reply to the terminal. This reduces the chance of other applications waiting for locked pages.

5.7 PSB Replacement

A DL/I database PCB defines an application's view of the database. A PCB defines which of the segments in a database the program is allowed to access and how the program is allowed to process the segments it accesses.

The extent to which the capabilities of the PCB have been used will affect the scope of the migration effort. The PCB's encountered may access only a few segments in a database or they may be general in scope and allow access to the entire database. To document the migration requirements, each PCB must be analyzed for the data access functions provided. In turn, each end user or program reference to the PCB must be analyzed for the data access functions actually required.

In general, the functions provided by a DL/I PCB will translate into SQL/DS views. Some general guidelines for translating PCB's to SQL/DS views are:

- As a general rule, always use SQL/DS views rather than base tables for database access.

---

[3] A DL/I application that needs to use a secondary index to reach the data must be coded to take advantage of the alternative path. This is not true with SQL/DS as the optimizer chooses the path and the application will work, even with no index (though performance may suffer!). There may be a tendency to create extra indexes during a migration because it makes access to particular data easier. However, there are costs associated with maintaining indexes. Each extra index will have to be maintained during a row insert or delete, and probably during a row update. They can also increase deadlocks and timeouts. It is suggested that the need for extra indexes should be reconsidered during the review.

- Define one view for every SQL/DS base table that is a full subset of that table.
- For small specialized PCB's, create one SQL/DS view to replace the PCB.
- For large, general purpose PCB's use the full subset views of the corresponding SQL/DS tables for database access rather than creating a new set of views specifically for the PCB. Exceptions may be identified where a specialized SQL/DS view would be more suitable.

A strategy for identifying the necessary views should be part of the migration plan.

5.7.1 Views of Data

Data access can be defined with SQL/DS views to limit access or to simplify the coding requirements. A DL/I PCB could correspond directly to an SQL/DS view. Special attention should be paid to situations where the update of more than one SQL/DS table is required in order to duplicate the PCB view of the database.

The use of SQL/DS views to control access is beneficial depending on the requirements of each use. In some cases a DL/I PCB view may be implemented more efficiently by specifying a subset of columns in the SQL statement rather than defining a separate SQL/DS view. Also, the benefit of defining a view as a means of granting access privileges on a restricted subset of data may be diminished if the view is used only in a program that is preprocessed into a static access module.

5.7.2 Access Authorization

Whether access is requested through SQL/DS views or the actual base tables, the necessary authorization is defined by the SQL/DS privileges in effect for the associated user. Note that these privileges are granted independently of any access request, and the resultant privilege set is what is used to authorize each request as it occurs.

Where multiple DL/I PCB's have been defined just to satisfy different sets of security requirements, those PCB's may be resolved to a common set of SQL/DS tables or views.

5.8 Database Design Checklist

Below is a summarized list of database design considerations for migrating DL/I databases to SQL/DS.

- Have **ALL** the original segments been accounted for?
- Have **ALL** the original attributes/ fields been accounted for?
- Has each SQL/DS table a (unique) primary key?
- Where data is used as foreign keys for joins, is it defined exactly the same in each table?
- Has SQL/DS referential integrity been used sensibly and have any occurrences where it was not transferable been documented and included in the application?

- Has the Outer Join possibility been considered and action taken? (Path calls for DL/I may not give the same results as an SQL/DS join if a row in one table is missing.)
- Are all columns NOT NULL? (Nulls are probably not applicable to migrated data, except possibly for an XDFLD with NULLVAL or EXTRTN.)
- Have **ALL** the applications' reliances on DL/I structures (e.g. for GNP's) been reworked?
- Have searches been made for use of "dirty" data, e.g. data which has a special meaning for an application, or data which varies in use according to a key value?
- Have **ALL** DL/I exit routines been reconsidered for SQL/DS?
- Have "code tables" been handled sensibly? (Many small code tables in SQL/DS may result in many unneeded SQL calls.)
- Have any history tables been carefully considered?
- Do any tables have a large number (4+) of indexes?
- Have the critical and/or potential problem transactions been identified?
- Have estimates been made for transactions, batch and utility runs?
- Has EXPLAIN[4] been used to confirm paths selected with 'real' values inserted into the catalog of the test system?

---

[4] SQL/DS EXPLAIN provides information about the access paths available and which path was chosen by the optimizer for the SQL call. It is used to ensure that where indexes have been defined to speed access to the data, that they are actually being used.

EXPLAIN is used in development but sometimes the small volumes of data in a development system can give a false picture of the path that will be selected during production. Catalog statistics can be updated, and it is vital that if system testing does not use data of similar volumes and makeup to production, that these statistics are updated to reflect the production values before programs are prepared and EXPLAIN is used.

Chapter 6. Data Migration

This chapter covers the issues of migrating data from existing DL/I databases to SQL/DS tables. Also, we discuss the issues of loading data into SQL/DS tables and the process of removing data inconsistencies.

The data migration process follows a sequence of steps, some of which are repetitive.

The suggested steps in the process of data migration are as follows:

1. Create or select the migration programs for extracting the data from the DL/I databases.
2. Run the extraction programs as many times as necessary to extract the data for each of the tables. All of the data should be extracted for all of the tables in one pass of the DL/I database, if possible. The output of the extract should be sequential files with the data in the layout of the rows for each table. This is the ideal. It may not be possible to unload the data into the row layouts on the extraction pass of the DL/I database. It also may not be possible to only make one pass of the DL/I database.
3. Process the sequential files from the extraction runs and clean up the data, making data conversions, if required, and formatting the data for loading into the SQL/DS tables.
4. Load the data into the SQL/DS tables using the SQL/DS Database Services Utility (DBSU) DATALOAD function.
5. Complete the cleanup of the data, if necessary, using (for example) ISQL or QMF.

We will discuss these steps in more detail in the balance of this chapter.

In some instances, because of use of data by applications being migrated and also by applications which are to remain for some time in DL/I, it will be necessary to have replicated data. That is, data will exist both in its original form in the DL/I database and in SQL/DS tables. With multiple copies of the same data, it will be necessary to consider how the two copies will be kept consistent. Whenever an update is made to either database, the update must be propagated to the other in some fashion. This is an issue that is covered in detail in Chapter 10, "Coexistence Strategies" on page 161.

6.1 DL/I Database Unload

It might be possible to read data from the DL/I database and insert it directly into SQL/DS tables in the same program. Several considerations suggest this may not be a good approach.

- The process of building primary and foreign keys can result in complex access patterns that are better accomplished by multiple extraction passes of the data.
- Referential constraints defined for the data may result in requirements for loading tables in a sequence that may not be practical in a single program.

- Better performance can probably be achieved by extracting the DL/I data to an intermediate file, sorting it into an appropriate sequence for SQL/DS, and then using the DBSU DATALOAD function.

The single program approach does not usually appear to be a workable approach and will not be discussed further. The use of intermediate files appears to be the best approach for data extraction.

Intermediate extraction files can be of several types. If the program can extract the data directly into the format of the SQL/DS row, a separate intermediate file may be generated for each of the tables. In some cases, it may be more appropriate to do record formatting in separate programs (see discussions of specific situations below). In these cases, intermediate files will be designed to meet the specific needs for formatting and sorting the data.

6.1.1 Considerations

There are a number of considerations for developing data conversion procedures. The primary concerns in the development of a set of conversion programs are:

- Ensure full integrity of the data loaded into SQL/DS tables.
- Extract all data required to load into the SQL/DS tables. (Manual intervention should be avoided).
- Convert data formats based upon the logical design of the SQL/DS tables. This may require application logic.
- Minimize execution time of the extracts. This means minimization of random accesses of the DL/I database and the number of passes over the database.

Data Coexistence

Careful consideration should be made before making the DL/I data available for access by DL/I applications while extraction is taking place. At least, access to the DL/I database should be restricted to inquiry only. It would be extremely difficult to get the data consistent in both the DL/I database and the SQL/DS tables if updates to DL/I databases were not prohibited during the extraction process and subsequent load of the SQL/DS tables.

Extract Source

It is assumed that the extract programs will have to operate against a static version of the database. It is also assumed that it may be necessary to run the extract several times while verifying its function, building test databases, and going through cutover rehearsals. This means that production must stop for the duration that the database is active during extract runs. To minimize the time that the database is unavailable while testing the extraction procedures and programs, consider generating a copy of the database. This requires additional disk space, but frees the database to resume production in the shortest time possible.

Gathering the data to build the record required for input to the SQL/DS DATALOAD utility may require accessing several segment types to derive keys. This may be done by directly accessing all required segments in a single pass, or with multiple passes over the DL/I database for a single segment type. The latter case would require sorts and merges to construct the final load input.

Approaches

There are three approaches for performing data extraction:

1. Use existing logical unload (extraction) utilities, either customer or vendor developed.
2. Write a program (or series of programs) specific to the task of data conversion.
3. Use the output of the DL/I unload utilities.

The first is the simplest, but may be too slow when working with large databases. The second has the potential for good performance, but could be expensive to write. The third requires knowledge of the format of the unloaded data. We will discuss the three approaches below.

6.1.2 Logical Unload Utilities

DL/I installations may have developed their own programs for the logical unload of their databases. If so, they have probably been designed to get the best performance possible with the structure that exists. These programs make a good base on which to build the logical unload that is required for the migration. Several passes of the database will be required when building primary and foreign keys.

If there is no logical unload program, it will be necessary to write a program or use an extractor to provide the data in a format useable by the SQL/DS DBSU DATALOAD function.

6.1.3 Write a Data Conversion Program

If it has been decided that a program or set of programs is the best solution, there are many ways to accomplish this. Only two are addressed here in detail. The first is to write a program which operates directly against the DL/I database to obtain all of the data necessary to build a record which can then be passed to SQL/DS. The second is to utilize the DL/I unloaded record format to generate a series of extracts, sorts, merges and conversions that produce the same result.

Considering the design of the unload program from the objectives of the program, record level issues, inter-record issues, and element/column issues, there are several alternatives to any approach. The following section describes at least one way of handling each of the issues.

Segment-to-Table Mapping

If the decision is made to make a simple segment to row conversion, data can be extracted and loaded with minimal effort. The more differences that exist between the two database designs, the more effort that will be required for an intelligent extraction and reformatting of the rows into tables.

Splitting segments: In the case of repeating groups or some cases of redefines, the DL/I segments will be divided into multiple rows. In this case, the primary key will be reproduced in each of the rows. Typically, there will be two types of records generated; one containing the basic single occurring attributes from the segment, and a second containing the repeating information. In the case of the latter, if a new primary key has been defined during design for this new segment/table, it must be constructed during the extraction process.

Combining Segments: When logical design dictates that the data in multiple segment types be combined into the same table under a common primary key, the data conversion programs must combine the data and insure that the primary key is included in the resulting rows. Most likely the source segments will not contain the primary key as a data field. The data conversion programs will need to merge the two source segments and add the primary key to the merged data. A sort can be used to merge the data from the source segments provided a common control field can be found in both segments.

The merge process is not always straightforward. It must be determined whether it is acceptable to permit unmatched conditions. If so, the fields of the missing segment should be set to NULL or other value indicating missing information.

There is also the possibility of the same attributes existing in more than one segment type. The attribute from one of the segments should be accepted and the other attribute ignored. The date of last update information in a segment may be available to determine whether the content of one segment is more recent than the other.

In those cases where complete accuracy is mandatory and the correctness of the individual segment types is not clear, intervention may be required. As the number of inconsistencies should be small, exception lists will normally be adequate to address the inconsistencies. In those cases where large volumes are anticipated, an interactive procedure can be developed to examine and repair the exceptions. By writing exceptions to an SQL/DS table, ISQL or QMF (for example) can be used for review of the exceptions and repair of the primary table.

Key Construction

The most complex requirement is that of constructing the primary and foreign keys necessary for SQL/DS referential integrity and for translation of the structural relationships in the DL/I database. The logical design of the database will have identified what keys are required and where the values are to be found. This information should be documented during the table design phase. The task of the conversion programs is to obtain these keys in the least costly way. Unfortunately, there seems to be no easy, high performance technique for doing this.

Primary Key: The primary key data can be found in the root segment for a root segment table or in the key feedback area for a dependent segment table. These keys can be obtained with the segment data by a series of passes of the database, retrieving just root segments for the primary key for a root segment table, or performing path calls to retrieve the concatenated key for dependent segment tables. This can be an extremely long process for large databases.

In the case where dependent segments do not have a sequence field, the method developed in the logical design for creating unique keys will need to be incorporated in the logic of the extraction programs.

Where timestamps have been defined as keys to provide for chronological sequencing, it will not be possible to generate a meaningful timestamp during conversion. Current timestamps may be used with care so that they are assigned to the records in the correct sequence.

Foreign Key: While primary keys may normally be built from data existing within the segments, foreign keys may require random access to obtain the data required to create the keys.

6.1.4 DL/I Unload Utility File

The DL/I unload utilities for HISAM and HD databases produce an intermediate file with both data content and structural relations information. Documentation of the content and description of this intermediate file is available. If the person developing the conversion unload function is familiar with the format of that file, it can be used as the base for data conversion. Using this file has the double advantage of providing an information source independent of the production database and providing the database structural information as well as the data content.

The unload file can be used as input to a user-written program that can retrieve the segment data, and format it for input to the SQL/DS DBSU DATALOAD function. The primary and foreign keys can be developed and appended to the data for each row. It may be possible to read the unload file once and create output files that can be used to load all of the SQL/DS tables that represent the DL/I database. This would be desirable because of the time required to read the unload file, but this objective should not result in an extract program that is so complex that it requires excessive resources to develop.

Internal Mapping

REDEFINES: Where REDEFINES exist, the logical design decisions must now be implemented in the conversion programs. The logic of the application that determines which data format is present must be provided in the conversion programs in order to allow movement of the data into the proper columns. Unused columns in any record should be set to NULL or other values indicating missing or inapplicable information. For those cases where multiple record types are to be generated, see "Splitting segments" on page 79.

Incompatible Formats: Cases where the DL/I data format is incompatible with available SQL/DS data formats must be taken into account in the conversion process. Data conversions will need to be made. Group level items must be broken into individual columns, combined as a single column, or combinations as dictated by the logical design.

Data Volume to be Converted

The volume issues here are threefold:

- Is there a comprehensive test database for integrated test?
- Can the entire database be used during the testing phases?
- How long will it take to convert the desired data?

Depending on the answers to these questions it may be necessary to develop a conversion technique for a test database as well as for the complete database.

from the DL/I database onto intermediate files. The files must then be matched with mismatches written to an error file (or table). Enough identifying information must be with each copy to permit analysis, probably interactively. A default may be taken to allow the initial loading of the tables (e.g., use the value from the record with the most frequent update cycle) and allow exceptions to be handled manually with a tool like ISQL or QMF.

- When segments are combined to make a single table, intermediate files will have to be merged unless they are extracted together. Some action will have to be taken for unmatched cases. These will probably be written to an error file (or SQL/DS table) for interactive evaluation. Whether it is logical to generate a partial row from the information that is present will be a matter to be resolved with the logic of the application.

DATALOAD control statements must be created. These map the input file fields to the SQL/DS columns. One set of statements will be required for each SQL/DS table. The loads may be run after deactivating the primary keys of all parent tables if deferred enforcement of referential constraints is desirable. The primary keys must then be activated after the loads are complete to verify the referential constraints of the tables.

Final preparation for execution such as invocation of ARCHIVE and UPDATE STATISTICS can now be done. The requirements here are the same as those for a new set of tables being added to an SQL/DS installation.

6.3 Data Conversion Validation

It is necessary to validate that the data has been successfully converted from the DL/I database to the SQL/DS tables. Many ways exist to do this and the best way will depend on installation requirements and available programs.

If audit programs already exist to validate database content, these can be used to validate the converted data. Such programs will need to be converted at any rate as a part of the conversion process. Once converted, programs that develop and check record counts and hash totals should work against the SQL/DS tables.

Many applications have reporting programs that read all records of a type and produce reports with totals for application balancing. These reporting programs can provide an important part of validation of the database conversion by comparing output of the program run against the DL/I version to output run against the SQL/DS version. A programmed comparison of the reports generated by the DL/I system and the ones generated by the SQL/DS system will uncover both program and conversion errors.

In those cases where there are no application programs to validate the database conversion, it will be necessary to provide another means of verifying the database content. There are a number of methods available. DL/I numeric fields can be totaled as hash totals and compared to the corresponding SQL/DS data hash totals. Alphabetic fields are difficult to verify, but typically less critical. Comparison of reports as a normal part of the testing process should provide adequate validation of these columns.

Chapter 7. Application Program Migration

7.1 Introduction to the Migration of Application Programs

This chapter discusses various aspects of migrating application programs that operate with DL/I (hierarchical) data to programs that operate with relational data. It gives you some idea of what lies behind the processes involved, general considerations, migration of DL/I calls[5] or commands to SQL implementations, and some special cases.

Migration of application programs occurs after it has been determined that some or all required data will be maintained in a relational database. Applications can be migrated partially or completely to a relational database depending on whether the data used by the application is migrated partially or completely.

Your current application may be utilizing DL/I hierarchical databases and/or flat files. Modifications to the application may include some combination of:

- Adding SQL to access new relational tables
- Changing existing DL/I calls[5] to logically equivalent SQL statements
- Changing existing "flat file" calls to SQL statements
- Eliminating existing DL/I or "flat file" calls
- Replacing the application with procedures that exploit ad-hoc capabilities of the relational system.

When you consider coexistence, migration, and the timing of each, there is a whole spectrum of possibilities. Coexistence strategies are discussed in greater detail in Chapter 10, "Coexistence Strategies." This chapter will focus on those portions of an application that need to change, with the overall guideline of changing only that for which change is essential.

7.2 Overview of the Process

In general, the process of migrating an application program consists of the following steps:

1. Determine which programs should be migrated.
2. Analyze the existing DL/I application program.

   This provides you with information that you use in Steps 3 and 4.
3. Replace DL/I interface specifications.

---

[5] Two DL/I interfaces are available for use in writing application programs: the DL/I High Level Programming Interface and the DL/I CALL interface. These two interfaces have different syntax and different names (DL/I command and DL/I call respectively) but perform the same DL/I functions. Throughout this chapter the term DL/I call is used to include whichever interface the application program is written in.

4. Change the DL/I calls[5] to appropriate SQL statements.

   New definitions are required, and you can use tools to generate the statements. Table and cursor definitions are added as required to the application programs. See "Use of Cursors" on page 87 for more information.

5. Modify error routines (severe and message) and date-handling routines that are currently implemented.

6. Assuming the database definitions and data have been migrated into the relational database system, test your programs. (This is often initially done with test data, which could be a subset of data that is used in production.)

These steps are discussed further in the following pages and in greater detail following 7.3.1, "Analysis of DL/I Calls or DL/I Commands."

7.2.1 Programs to Be Migrated

You should use the following criteria when determining which programs should be migrated. Certain programs may no longer be required because:

- They can be replaced by an ISQL or a QMF report or query
- They are made obsolete because of the new database design
- They are no longer applicable to the relational database management environment (for example, specific load functions).

7.2.2 Analyzing Existing DL/I Application Programs

Before you can begin the migration of a program you should analyze it to obtain some specific data about the program. This includes information such as the function of the program, the number and types of DL/I calls in it, and which databases and segments it uses.

This is important information about the program as it helps considerably in determining how DL/I calls are to be changed to SQL statements.

This analysis enables you to determine:

- Which calls can be dropped from the DL/I program because they are redundant or not necessary in SQL.

  For example, positioning calls (see "Positioning Calls") may no longer be needed.

- How the number of calls can be reduced.

  For example, a function performed by several calls in the DL/I program may be performed by one call in the relational application program.

- How program logic (related to the DL/I calls) can be simplified.

Positioning Calls

A DL/I application program uses a positioning call to access data in lower levels of hierarchical data structures, starting from a particular point. A positioning call is also required for use with DL/I DLET or REPL operations. However, with SQL you have the capability to go directly to all data. This means that positioning calls are no longer required for this function within a migrated program.

Here is an example of what you might do when removing these calls during application migration. The example is of two positioning calls in a DL/I application program:

```
GU Call to Segment SEGA
          .
          .
(No processing or data analysis of SEGA)
          .
          .
GNP Call to Segment SEGB
```

In the relational database system these two segments may have been replaced by two corresponding tables. Now, because you can read the data directly from the table corresponding to SEGB that has been created in the relational database, you can remove the GU call completely. However, there must be a relationship between the two tables. For this purpose, the table contains a ***foreign key***. This foreign key is the key from the SEGA segment.

Where you find positioning calls in the application program you are migrating, check the new table structures for the segments affected and ensure the correct foreign key connections are made.

Sometimes, positioning calls are also used to provide dedicated error messages. For example, 'SEGA key not found' rather than 'SEGB key not found'. In such cases you must decide if the error message is still needed (wanted). If yes, the call may be kept, otherwise it can be removed. (Alternatively, testing for existence of a particular value of the primary key of SEGA may now be done via the table corresponding to SEGB, since that table now includes that data as a foreign key. The application need not actually access the table corresponding to SEGA, if the application may thereby be simplified.)

Use of Cursors

The method of retrieving multiple rows in a relational database table involves the use of a named control structure called a *cursor*. This method means that cursors must be added where required to your application programs when migrating. If the application logic requires retrieving multiple rows **in a given sequence** (which is typical in many DL/I applications) then an SQL ORDER BY statement must be included in the cursor declaration.

There are some particular instances when cursors should be used. When converting an application program you must:

1. Use a cursor when, as a result of a loop including a GET call in the DL/I program, you could possibly retrieve multiple segments. The cursor is then used to identify the particular rows of interest. Each row, which meets the selection criteria (WHERE clause), can be read using a FETCH statement.
2. Use a cursor for update when a GET HOLD type call has been replaced by a FETCH statement that is followed by an UPDATE or DELETE statement **and the sequence of retrieval is not required by the application logic**; that is there is no SQL ORDER BY in the cursor.

In the converted program the cursor must be:

1. Declared using DECLARE CURSOR

2. Opened using OPEN CURSOR (after the WHERE clause and ORDER BY variables are specified)
3. Referred to in a FETCH statement and possibly in UPDATE and/or DELETE statements with the WHERE CURRENT OF clause.

   Note: The UPDATE and/or DELETE statements with the WHERE CURRENT OF clause cannot be used if the result table of the cursor is read-only, e.g. the cursor contains an ORDER BY statement.
4. Then closed using CLOSE CURSOR.

See 7.3.2, "Changing DL/I Calls or DL/I Commands To SQL Statements" on page 93 for examples of the implementation of cursors in application programs.

Aggregations or Built-In Functions

The SELECT statement permits several built-in functions or aggregations. You can use these functions in an application program and in this way reduce your own programming as well as improve performance.

If the DL/I program is using logic to count, average, etc., you may be able to eliminate some code by migrating to SQL. The number of I/O's to accomplish a task is a significant cost factor in any application. The number of rows actually retrieved into the application program's data areas is also a significant cost factor. By using aggregations, this number may be greatly reduced. It is cheaper to use an aggregation than to retrieve all rows into the application's data areas and let the application perform the aggregation.

Refer to *SQL/DS Application Programming* (VM: SH09-8086, VSE: SH09-8098) for further information regarding these built-in functions.

7.2.3 Replacement of DL/I Interface Specifications

All parts of existing application programs that relate to DL/I databases that are being migrated to SQL/DS must be removed or replaced,

You can remove:

- SSAs[6] (but keep search fields as host variables)
- PSB name and definitions
- User Interface Block (UIB) (for CICS-DL/I environments)
- Base Linkage Locators (BLL) for the PCB addressing and the PCB-pointer addressing
- PCBs and PCB pointer
- Checking of the User Interface Block (CICS-DL/I environments)
- Definitions of the database-call functions
- Tests within online CICS programs for errors pertaining uniquely to CICS-DL/I
- Scheduling and termination calls or commands.

---

[6] The DL/I High Level Language Interface does not use Segment Search Arguments. This function is supplied with a SEGMENT and WHERE clause in the DL/I command. The same function is supplied as with the SSA in the DL/I Call interface but it is coded differently in the application program.

You should replace:

- DL/I I/O areas with host variables.
- DL/I calls or commands with SQL statements. (This should be performed selectively in each program.)
- Database severe-error processing with new routines (including new definitions).
- Moves in the SSA[6] search fields with moves in the corresponding fields of host variables.
- Segment I/O areas, initialized with default values, with working storage variables initialized with default values, which are moved into the host variables before a new row is inserted.

7.2.4 Converting the DL/I Calls or DL/I Commands.

The DL/I calls or DL/I commands in the application program have to be changed into the appropriate SQL statements.

This necessitates the addition of table and cursor definitions to the application programs. These statements can be generated if you are using a product such as IBM's DBRAD (Database Relational Application Directory).

You should SELECT only the required columns (fields). Remove the DL/I calls not required in SQL and restructure the application programs accordingly.

There is a detailed explanation of how to do this, and many examples, following 7.3.1, "Analysis of DL/I Calls or DL/I Commands" on page 93.

Note: You may use SQL statements and DL/I calls or commands within a single application program. This does not create integrity problems for online programs under CICS. A batch program, however, should be restricted to updating either DL/I data or SQL/DS data but not both. This is true for DL/I batch and MPS (For more details refer to Chapter 10, "Coexistence Strategies" on page 161).

7.2.5 Error Handling

In DL/I, each call or command has a resulting status code (or return code). New code must be developed in the relational database application program that performs a similar operation. A comparison of the DL/I status codes against the corresponding relational database return codes must be made.

Error message handling routines and severe error routines for batch and online programs may need to be coded. However, many installations have one or a few error routines used by all applications. Though modification of such a routine to (also) handle SQL error codes may be a significant task, once it is done, the resulting code may be shared by many or all applications.

For more detailed information refer to 7.4.7, "Error Handling" on page 131.

7.2.6 Date Handling

Routines to convert the date into the appropriate formats may be required. Such routines should enable you to edit the date from an external format to an internal format and vice versa.

It is recommended that you use the data type DATE in the relational database tables, but continue to use the current packed date format in the application program. You may consider changing all date formats to:

```
PIC S9(7) COMP-3
```

There are practical programming reasons for doing this, and it should also help to minimize the conversion effort required.

For more detailed information refer to "Date-Field Handling" on page 128.

Note: Date conversion also gives you an opportunity to address the problems created by the year 2000 (that is, the century change). However, this does require additional conversion effort.

7.2.7 Other Considerations

There are several other considerations that have to be taken into account when developing the relational database application programs. Primarily these are due to the different way certain aspects are implemented in the relational database management system. A brief description of these considerations follows.

Field Initialization

In many cases DL/I installations use copybooks and initialization I/O areas for putting data into fields. You may consider providing copybooks that move the required default values into the host variables.

For more detailed information refer to 7.5.3, "Initialization of Fields (Defaults)" on page 136.

Field-Level Programming

The relational database system enables you to read information in individual columns of a table. This was not possible in DL/I where the complete segment had to be read.

Field-level programming can simplify and speed up the execution of the application program, but you should carefully examine which data you use.

The most important aspect of field-level programming is that your application programs become less dependent on the physical storage of data. This means that database changes have less impact on your application programs.

For more detailed information refer to 7.5.4, "Field-Level Programming" on page 136.

Search Keys

The relational database system provides a more easily understood method (using foreign keys) of providing logical links between data. This method must be considered carefully in the design of the application programs to be migrated.

It may be advantageous to remove program logic that ensures the consistency of data, if you are using the referential integrity facilities of the relational database management system.

For more detailed information refer to "Setting of Search Keys" on page 114.

7.2.8 Program Preparation and Execution (Testing)

The following figure graphically shows how an application program is prepared and executed.

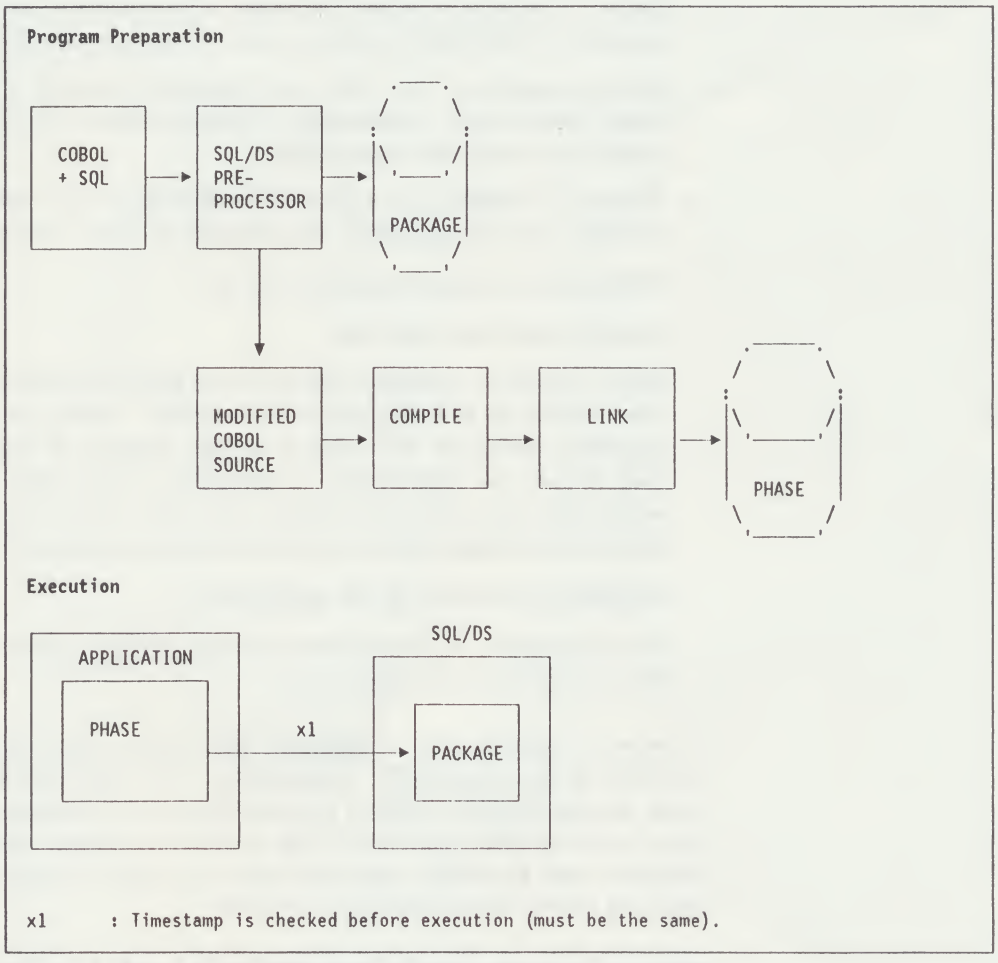


Figure 3. Steps in SQL/DS program preparation and execution.

Once the SQL/DS program has been prepared, then you are ready for testing. The proper data must have been migrated into the SQL/DS tables prior to testing. Refer to 7.6, "Unit Testing" for further detail.

7.3 Program Migration Strategy

As an overall program migration strategy, the following points should be considered:

- Keep changes to a minimum.

  If you can reuse the code, do so. If a portion of the application can be eliminated, do so. If modules can be made common and reused in various applications, then create those common structures. If an ISQL or a QMF

ad-hoc query can replace a program, then try to use such flexible options. Of course, performance is a consideration, but migration time can often be greatly reduced by using saved queries rather than recoding an application. A QMF query can be run as a batch report (under VM) with comparative ease.

- Determine naming conventions.

  Usually, existing naming conventions are adequate for coexistence. When migrating an application, it is a good time to evaluate your overall naming convention to ensure that all foreseeable requirements can be accommodated. Consider whether the application may be used in a distributed environment in the future; naming conventions should allow for such conditions.

  During migration, you also may decide to enlarge data element names to make them more meaningful. Consideration may be given to the use of CASE tools and their capabilities.

- Develop a strategy for migration of functions that must be implemented differently (such as scrolling - see details in 7.4.8, "Scrolling").

- Establish parameter passing methods.

- Analyze common modules.

  Often, common modules that exist for the DL/I environment may be directly convertible to the SQL/DS environment. There may be some common modules, however, that are no longer needed in the SQL/DS environment and should be eliminated. Conversely, there may be common modules which did not exist, but which you will want to create and add to your library to facilitate future development, conversions, and maintenance.

- Migrate the balance of the application.

  The remainder of this section provides further detail regarding migration of the bulk of the application code.

In order to migrate any application, you should get a general overview of the functions of the application program(s). This may be obtained from the code itself, documentation, original programmer, or the end user. It is best to understand what the end user thinks the program is doing, since not all parts of the program may be easily migrated and you want to ensure that more complex changes still produce the desired results.

When you want to migrate a program, you need:

- Host variables for all table columns used by the program.

  These could be defined in copybooks or modules that are brought into the program as required, yet would be common for your operating environment.

- SQL statements corresponding to the processing requirements of the application program.

  Much of this chapter is devoted to details of converting various DL/I calls to appropriate SQL calls.

- Authorization on all base tables and/or views used by the application. Recommendations regarding use of views are discussed in the section "Data Security" in Chapter 5, "Database Design."

7.3.1 Analysis of DL/I Calls or DL/I Commands

Every application program containing DL/I calls or commands must be analyzed to collect the following information:

- Types of DL/I call or command (GU, GHU, ISRT, etc.).
- Was the call function a variable or is it a constant in the SSA[6] ?
- Which database and which segments are processed?
- Usage of SSAs[6] (qualification).
- Which keys are used and what are the key fields?
- Was the compare operator modified?
- Does the call use the *parm-count* parameter as part of the argument list and is the *parm-count* used as a variable?

If a program contains copybooks with SQL statements, these can no longer be copied with the COPY verb into the application program. They must be included into the application program with the statement:

```
EXEC SQL INCLUDE membername END-EXEC.
```

Note: A member included into the program by the EXEC SQL INCLUDE statement cannot contain further EXEC SQL INCLUDE statements.

7.3.2 Changing DL/I Calls or DL/I Commands To SQL Statements

Once you have identified (and understood) the individual DL/I calls or commands in the application program you are migrating, change the calls or commands to functionally equivalent SQL statements.

The conversion of the following types of DL/I calls or commands is discussed:

- **The DL/I calls discussed are:**

  Get Unique (GU) Call

  Get Hold Unique (GHU) Call

  Get Next (GN) Call

  Get Hold Next (GHN) Call

  Get Next within Parent (GNP) Call

  Get Hold Next within Parent (GHNP) Call

  Path Call

  Delete (DLET) Call

  Replace (REPL) Call

  Insert (ISRT) Call.

  Note: There is a functional similarity between GET calls with HOLD and the same calls without HOLD, which allows the programmer to group these calls together functionally. The only difference between these calls is that the HOLD part of the call allows the programmer to issue a subsequent REPLACE or a DELETE call (using the same PCB). If the GET HOLD is not followed directly by a REPLACE or a DELETE call, using the same PCB, the

hold status is lost, and it must be reestablished by another GET HOLD before a REPLACE or DELETE can be processed.

- **The DL/I commands discussed are:**

  GET UNIQUE (GU) Command

  GET NEXT (GN) Command

  GET NEXT IN PARENT (GNP) Command

  Path Call

  DELETE (DLET) Command

  REPLACE (REPL) Command

  INSERT (ISRT) Command

  Note: The DL/I Application Programming High Level Programming Interface does not have any HOLD type commands. The developers of the interface decided to convert all GET type commands into internal GET HOLD type calls therefore simplifying the programmers task.

The description of the DL/I calls or commands given here consists of:

A summary of the call or commands's function

The associated Segment Search Arguments (SSA)[6] if the DL/I Call interface is being used or

The SEGMENT selection and WHERE phrase if the DL/I Command interface is being used

The functionally equivalent SQL statement and programming.

Note: The SQL implementation described here is only one method of converting DL/I calls or commands; other possibilities exist. You should explore development of your own methods as alternatives to the methods suggested here.

7.4 Program Migration: Detailed Recommendations

This section addresses the major considerations for converting an application from DL/I to SQL/DS. While this is not an exhaustive list, the items covered here include:

1. Navigation Commands
2. Data Modification
3. Referential Integrity
4. Set Processing
5. Special Cases: Dates, Times, and Fillers
6. Column Selection vs. Record Selection
7. Error Processing
8. Scrolling

7.4.1 Navigation Commands

We will now discuss migration of navigational commands in DL/I and SQL/DS. These DL/I calls or commands are used to establish parentage within the database hierarchy. Such positioning calls or commands are generally not needed in the SQL/DS application. There is some relationship to the SQL/DS WHERE clause, especially when a foreign key has been defined.

A path call may be issued in DL/I to pull data from multiple segments. Within SQL/DS, this may require JOIN statements or multiple SQL statements. Included in this section is a discussion and example regarding setting of the search keys for positioning and the associated method used in an SQL/DS application to accomplish positioning among multiple tables. But, first let's examine the GET calls or commands which establish position.

Note: The following topics discuss the DL/I Call and DL/I Command interfaces separately. This leads to some redundancy but is easier to follow because the examples are quite different in syntax. The reader should read the individual sections (Calls or Commands) that apply to the DL/I interface in which the application being converted is written.

Converting the Get Unique (GU) and Get Hold Unique (GHU) Calls

Note: If the application program you are converting uses the DL/I Command interface, please skip this section and go to the section titled **"Converting the Get Unique (GU) Commands" on page 100** .

The GU and GHU calls randomly retrieve any sensitive segment in the database depending on the SSAs defined and the PCB used. They are also used to establish a position in the database for subsequent GN calls, REPL (replace) calls, or DLET (delete) calls, and to establish parentage for GNP (Get Next within Parent) calls.

The following SSA definitions are possible:

- No SSAs.

  If no SSAs are defined the call retrieves the first (root) segment in the database.

- Any SSAs according to the hierarchical levels in the DL/I database.
- For one or more unqualified SSAs, the first segment found is retrieved.

  The first occurrence of the segment satisfies the unqualified SSA.

- If multiple SSAs are defined, the root segment identified by the first SSA is used as the parent root segment for whichever child is retrieved.
- Multiple qualified SSAs.

  The segment retrieved has the path of parents as defined by the SSAs (by segment name and keys).

- If one or more SSA's are omitted from a multiple SSA path definition, implied SSAs are generated depending on the previous call.

When converting these types of calls you must consider the segment names, the key fields used, the operator specified, and the key fields' content specified in the SSA.

GU Call without an SSA

This GU call is typically used only for positioning in the database. Check carefully the intent of the call and the associated application logic. You will need to rewrite the application logic to accommodate the set processing capability of SQL (see the following sections which discuss cursor processing). An example of this type of call is:

```
CALL 'CBLTDLI' USING func_gu,dbpcb,db_ioarea.
```

GU Call with a Single SSA and Operator '= ' or

The GU call, as shown in the following example, directly reads one segment:

```
CALL 'CBLTDLI' USING func_gu,dbpcb,db_ioarea,ssa1
```

Provided the key is unique, the conversion can be performed as follows:

```
MOVE SSA1-KEYFIELD TO HOSTVAR.

EXEC SQL
SELECT  ...,...,...  (Required field names)
INTO    ...,...,...  (Host variables)
FROM    VIEWname
WHERE   KEY-COLUMN = :HOSTVAR
END-EXEC.
```

GU Call with Multiple SSAs and Operator '= ' or

In the following example the DL/I call has three SSAs, which means the retrieved segment is specified in the last SSA. This means that the SQL SELECT should read the table that contains the data of the corresponding segment type.

The key field content of the three SSAs is used in the WHERE clause of the SQL SELECT statement.

If the DL/I call is:

```
                                        SSA referencing segment:

                                         ITEMID INVDAT  LOCATN
                                            |    |     |
CALL 'CBLTDLI' USING func_gu,dbpcb,db_ioarea,ssa1,ssa2,ssa3
                                            |    |     |
                      and                  "= " "= " "="  as operators
```

The corresponding code for SQL SELECT is:

```
MOVE SSA1-KEYFIELD TO HOSTVAR1.        The number of moves usually
MOVE SSA2-KEYFIELD TO HOSTVAR2.        corresponds to the number of
MOVE SSA3-KEYFIELD TO HOSTVAR3.        SSA key fields.  You can also
                                       use the SSA key fields directly
                                       as host variables in the
                                       SQL statement.

EXEC SQL
SELECT  colX,colY,...
INTO    ...,...,...
FROM    VIEWname
WHERE   KEY-COLUMN1 = :HOSTVAR1        Host variables 1, 2, and 3
AND     KEY-COLUMN2 = :HOSTVAR2        build the concatenated key.
AND     KEY-COLUMN3 = :HOSTVAR3
END-EXEC.
```

GU Call with SSA Operators Other Than '= '

As with the unqualified GU call which used no SSA's, this call is probably being used for positioning within the hierarchical database structure. But, after migrating a GU call with an SSA operator other than '=', it is likely that SQL will find more than one row that fits this specification. Therefore, this call must be converted with the help of cursors. If you do not use a cursor, SQL/DS returns an error SQLCODE (-810) (equivalent to SQLSTATE 21000) when more than one row meets the condition in the WHERE clause. Note that the DL/I program probably has loop logic to do GN after the GU positioning in order to retrieve multiple segments. That looping logic must be modified by using the FETCH statement as indicated in the following discussion.

The DL/I call is:

```
                                        SSA comprising segment:

                                         ITEMID INVDAT  LOCATN
                                            |    |    |
CALL 'CBLTDLI' USING func_gu,dbpcb,db_ioarea,ssa1,ssa2,ssa3
                                            |    |    |
                        and                "= " "= " ">=" as operators
```

Note: The last SSA has the operator '> ='.

A cursor must be used in the conversion.

The corresponding DECLARE CURSOR statement should be coded prior to the first reference to the cursor in the program:

```
EXEC SQL
DECLARE cursorname CURSOR FOR
SELECT  colX,colY,...
FROM    VIEWname
WHERE   KEY-COLUMN1 =  :HOSTVAR1
AND     KEY-COLUMN2 =  :HOSTVAR2
AND     KEY-COLUMN3 >= :HOSTVAR3
END-EXEC.
```

The following should be coded where the DL/I call was:

```
MOVE SSA1-KEYFIELD TO HOSTVAR1. Set the host variables used in the WHERE clause
MOVE SSA2-KEYFIELD TO HOSTVAR2. of the DECLARE cursor statement.
MOVE SSA3-KEYFIELD TO HOSTVAR3.

EXEC SQL                    SQL/DS logically materializes all data as
OPEN cursorname             defined in the cursor (SQL/DS internal).
END-EXEC.

EXEC SQL                    Loop to retrieve rows and store them
FETCH cursorname            in the host variables of the application program.
INTO  ..,...,...            See the explanation following.
END-EXEC.

EXEC SQL                    The cursor is no longer needed and is therefore
CLOSE cursorname            closed.
END-EXEC.
```

Because there may be more than one row that satisfies the SSA specification, several rows are retrieved by SQL/DS. One row is retrieved each time the FETCH loop is performed until there is no more data found. Then an appropriate return code (+100) is provided by SQL/DS.

Note: Make sure that you close the cursor before you open it again (may be with new key-field data).

Converting the Get Hold Unique (GHU) Call

How you convert the GHU call depends on the calls that immediately follow it. Different sets of circumstances can prevail:

1. If the GHU call is not immediately followed by a REPL or a DLET call, the GHU call can be converted exactly as a GU call.
2. If the call is followed by a DLET or REPL call, and the key qualified is unique, you can remove this call and perform just the respective UPDATE or DELETE. (See also "Converting the Delete (DLET) Call or Command" on page 117 and "Converting the Replace (REPL) Call or Command" on page 119 for more information.)
3. If the key qualified is not unique, then a set-level UPDATE or DELETE may be appropriate. However, if the conditions under which a record (row) is updated cannot be expressed in a WHERE clause but require programmed testing and analysis of row contents, then use a cursor-controlled UPDATE (with WHERE CURRENT OF clause).

   Note: SQL does not permit use of a cursor-controlled UPDATE or DELETE (with WHERE CURRENT OF clause) if the cursor contains a ORDER BY statement.

   As an example, the following DL/I calls (within a loop so as to process all segments meeting SSA conditions)

```
                                                SSA referencing segment:

                                                 ITEMID INVDAT  LOCATN
                                                   |    |     |
CALL 'CBLTDLI' USING func_ghu,dbpcb,db_ioarea,ssa1,ssa2,ssa3
                                                   |    |     |
                                                  "= " "= " "= " as operators
                                 and
                               or unqualified SSA

CALL 'CBLTDLI' USING func-repl,dbpcb,db_ioarea
```

could be replaced with the following:

In the program, prior to references to the cursor:

```
EXEC SQL
DECLARE cursorname CURSOR FOR
SELECT  colX,colY,colZ
FROM    VIEWname
WHERE   KEY-COLUMN1 = :HOSTVAR1
AND     KEY-COLUMN2 = :HOSTVAR2
AND     KEY-COLUMN3 = :HOSTVAR3
FOR UPDATE OF colY,colZ,colA
END-EXEC.
```

In the Procedure Division, where the DL/I call was:

```
MOVE SSA1-KEYFIELD TO HOSTVAR1. Set the host variables used in the WHERE clause
MOVE SSA2-KEYFIELD TO HOSTVAR2. of the DECLARE cursor statement.
MOVE SSA3-KEYFIELD TO HOSTVAR3.

EXEC SQL                    SQL/DS logically materializes all data
OPEN cursorname             as defined in the cursor (SQL/DS internal).
END-EXEC.

EXEC SQL          <---------+
FETCH cursorname            |  Loop to retrieve all rows and store them
INTO  ..,...,...            |  in the host variables of the application program.
END-EXEC.                   |  Update the row where necessary.
                            |
(Test: Should this row      |
   be updated?)  NO: -------+
                            |
(YES):                      |
EXEC SQL UPDATE VIEWname    |
  SET coly = :hvy,          |
       ... = ...            |
 WHERE CURRENT OF           |
              cursorname ---+

(No more rows to process):

EXEC SQL                    The cursor is no longer needed and is therefore
CLOSE cursorname            closed.
END-EXEC.
```

Converting the Get Unique (GU) Commands

Note: If the application program you are converting uses the DL/I Call interface, please skip this section and refer to the section titled **"Converting the Get Unique (GU) and Get Hold Unique (GHU) Calls" on page 95** .

The GU commands randomly retrieve any sensitive segment in the database depending on the SEGMENT selection phrase defined and the PCB used. They are also used to establish a position in the database for subsequent GN commands, REPL (replace) commands, or DLET (delete) commands, and to establish parentage for GNP (Get Next within Parent) commands.

The following SEGMENT selection definitions are possible:

- No SEGMENT phrase specified.

  At least one SEGMENT must be specified for GU commands. No SEGMENT phrases are required for GET NEXT or GET NEXT IN PARENT commands.

- Single SEGMENT phrase specified.

  Note: The lowest or only SEGMENT phrase is referred to as the object segment. If multiple SEGMENT phrases are used, all SEGMENT phrases above the lowest level are referred to as parent segments. The function of these parent segments is to identify the hierarchical path leading to the lowest level segment, the object segment.

- Multiple SEGMENT phrases according to the hierarchical levels in the DL/I database.

- For one or more unqualified SEGMENT phrases (no WHERE phrase), the first object segment found is retrieved.

  The first occurrence of the object segment satisfies the unqualified SEGMENT phrase.

- If multiple SEGMENT phrases are defined, the root segment identified by the first parent SEGMENT is used as the parent root segment for whichever child is retrieved.

- Multiple qualified SEGMENT phrases.

  The segment retrieved has the path of parents as defined by the SEGMENT phrases (by segment name and keys).

- If one or more SEGMENT phrases are omitted from a multiple SEGMENT path definition, implied SEGMENT phrases are generated depending on the previous command.

When converting these types of commands you must consider the segment names, the key fields used, the operator specified, and the key fields' content specified in the WHERE clause.

The DL/I command interface converts all GET commands to HOLD type calls internally. Therefore, how you convert the GU command depends on the commands that immediately follow it. Different sets of circumstances can prevail:

1. If the GU command is not immediately followed by a REPL or a DLET command, the GU command can be converted as shown beginning with **GU Command ...** on Page 102.

2. If the command is followed by a DLET or REPL command, and the key qualified is unique, you can remove this command and perform just the respective UPDATE or DELETE. (See also "Converting the Delete (DLET) Call or Command" on page 117 and "Converting the Replace (REPL) Call or Command" on page 119 for more information.)

3. If the key qualified is not unique, then a set-level UPDATE or DELETE may be appropriate. However, if the conditions under which a record (row) is updated cannot be expressed in a SQL WHERE clause but require programmed testing and analysis of row contents, then use a cursor-controlled UPDATE (with WHERE CURRENT OF clause).

   Note: SQL does not permit use of a cursor-controlled UPDATE or DELETE (with WHERE CURRENT OF clause) if the cursor contains a ORDER BY statement.

   As an example, the following DL/I commands (within a loop so as to process all segments meeting WHERE conditions)

```
EXECUTE DLI
     GET UNIQUE USING PCB(1)
     SEGMENT(seg-name) WHERE(field1-name = field1-reference)
     FIELDLENGTH(expression)
     SEGMENT(seg-name) WHERE(field2-name = field2-reference)
     FIELDLENGTH(expression)
     SEGMENT(seg-name) WHERE(field3-name = field3-reference) INTO(ioarea)
     FIELDLENGTH(expression) SEGLENGTH(expression)
END-EXEC.
      .        Program logic
      .                        Note that WHERE operators shown
      .                        are " = ", " = ", "="  but could
      .                        also be other values or the SEGMENT
      .                        phrase could be unqualified (WHERE omitted).
EXECUTE DLI
     REPLACE USING PCB(1)
     SEGMENT(seg-name) FROM(ioarea) SEGLENGTH(expression)
END-EXEC.
```

   could be replaced with the following:

   In the program, prior to references to the cursor:

```
EXEC SQL
DECLARE cursorname CURSOR FOR
SELECT  colX,colY,colZ
FROM    VIEWname
WHERE   KEY-COLUMN1 = :HOSTVAR1
AND     KEY-COLUMN2 = :HOSTVAR2
AND     KEY-COLUMN3 = :HOSTVAR3
FOR UPDATE OF colY,colZ,colA
END-EXEC.
```

   In the Procedure Division, where the DL/I call was:

```
MOVE FIELD1-REFERENCE TO HOSTVAR1.      Set the host variables
MOVE FIELD2-REFERENCE TO HOSTVAR2.      used in the WHERE clause
MOVE FIELD3-REFERENCE TO HOSTVAR3.      of the DECLARE cursor statement.

EXEC SQL                    SQL/DS logically materializes all data
OPEN cursorname             as defined in the cursor (SQL/DS internal).
END-EXEC.

EXEC SQL                    Loop to retrieve all rows and store them
FETCH cursorname            in the host variables of the application program.
INTO  ..,...,...            Update the row where necessary.
END-EXEC.

(Test: Should this row
   be updated?)  NO:

(YES):
EXEC SQL UPDATE VIEWname
  SET coly = :hvy,
       ... = ...
 WHERE CURRENT OF
              cursorname

(No more rows to process):

EXEC SQL                    The cursor is no longer needed
CLOSE cursorname            and is therefore closed.
END-EXEC.
```

GU Command with a Single SEGMENT and Operator '= '

This GU command is frequently used only for positioning in the database. Check carefully the intent of the call and the associated application logic. You will need to rewrite the application logic to accommodate the set processing capability of SQL (see the following sections which discuss cursor processing).

The GU command, as shown in the following example, directly reads one segment:

```
EXECUTE DLI
     GET UNIQUE USING PCB(1)
     SEGMENT(seg-name) WHERE(field-name = field-reference) INTO(ioarea)
     FIELDLENGTH(expression) SEGLENGTH(expression)
END-EXEC.
```

Provided the key is unique, the conversion can be performed as follows:

```
MOVE FIELD-REFERENCE TO HOSTVAR.

EXEC SQL
SELECT  ...,...,...  (Required field names)
INTO    ...,...,...  (Host variables)
FROM    VIEWname
WHERE   KEY-COLUMN = :HOSTVAR
END-EXEC.
```

GU Command with Multiple SEGMENT and Operator '= '

In the following example the DL/I command has three SEGMENT phrases which means the retrieved segment is specified in the last SEGMENT phrase. This means that the SQL SELECT should read the table that contains the data of the corresponding segment type.

```
EXECUTE DLI
     GET UNIQUE USING PCB(1)
     SEGMENT(seg-name) WHERE(field1-name = field1-reference)
     FIELDLENGTH(expression)
     SEGMENT(seg-name) WHERE(field2-name = field2-reference)
     FIELDLENGTH(expression)
     SEGMENT(seg-name) WHERE(field3-name = field3-reference) INTO(ioarea)
     FIELDLENGTH(expression) SEGLENGTH(expression)
END-EXEC.
```

The corresponding code for SQL SELECT is:

Note: The reference field content of the three SEGMENT WHERE clauses is used in the SQL WHERE clause of the SQL SELECT statement.

```
MOVE FIELD1-REFERENCE TO HOSTVAR1.      The number of moves usually
MOVE FIELD2-REFERENCE TO HOSTVAR2.      corresponds to the number of
MOVE FIELD3-REFERENCE TO HOSTVAR3.      WHERE clauses.  You can also
                                       use the WHERE field-references
                                       directly as host variables in
                                       the SQL statement.

EXEC SQL
SELECT  colX,colY,...
INTO    ...,...,...
FROM    VIEWname
WHERE   KEY-COLUMN1 = :HOSTVAR1       Host variables 1, 2, and 3
AND     KEY-COLUMN2 = :HOSTVAR2       build the concatenated key.
AND     KEY-COLUMN3 = :HOSTVAR3
END-EXEC.
```

GU Command with WHERE Operators Other Than '= '

This command is frequently used for positioning within the hierarchical database structure. After migrating a GU command with a WHERE operator other than '=', it is likely that SQL will find more than one row that fits this specification. Therefore, this command must be converted with the help of cursors. If you do not use a cursor, SQL/DS returns an error SQLCODE (-810) (equivalent to SQLSTATE 21000) when more than one row meets the condition in the SQL WHERE clause. Note that the DL/I program probably has loop logic to do GN after the GU positioning in order to retrieve multiple segments. That looping logic must be modified by using the FETCH statement as indicated in the following discussion.

The DL/I command is:

```
EXECUTE DLI
     GET UNIQUE USING PCB(1)
     SEGMENT(seg-name) WHERE(field1-name = field1-reference)
     FIELDLENGTH(expression)
     SEGMENT(seg-name) WHERE(field2-name = field2-reference)
     FIELDLENGTH(expression)
     SEGMENT(seg-name) WHERE(field3-name >= field3-reference)
     INTO(ioarea) FIELDLENGTH(expression) SEGLENGTH(expression)
END-EXEC.
```

Note: The last WHERE has the operator '> ='.

A cursor must be used in the conversion.

The corresponding DECLARE CURSOR statement should be coded prior to the first reference to the cursor in the program:

```
EXEC SQL
DECLARE cursorname CURSOR FOR
SELECT  colX,colY,...
FROM    VIEWname
WHERE   KEY-COLUMN1 =  :HOSTVAR1
AND     KEY-COLUMN2 =  :HOSTVAR2
AND     KEY-COLUMN3 >= :HOSTVAR3
END-EXEC.
```

The following should be coded where the DL/I command was:

```
MOVE FIELD1-REFERENCE TO HOSTVAR1.      Set the host variables used
                                        in the WHERE clause of
MOVE FIELD2-REFERENCE TO HOSTVAR2.      DECLARE cursor statement.
MOVE FIELD3-REFERENCE TO HOSTVAR3.

EXEC SQL                    SQL/DS logically materializes all data as
OPEN cursorname             defined in the cursor (SQL/DS internal).
END-EXEC.

EXEC SQL                    Loop to retrieve rows and store them
FETCH cursorname            in the host variables of the application program.
INTO  ..,...,...            See the explanation following.
END-EXEC.

EXEC SQL                    The cursor is no longer needed and is therefore
CLOSE cursorname            closed.
END-EXEC.
```

Because there may be more than one row that satisfies the DL/I WHERE specification, several rows are retrieved by SQL/DS. One row is retrieved each time the FETCH loop is performed until there is no more data found. Then an appropriate return code (+100) is provided by SQL/DS.

Note: Make sure that you close the cursor before you open it again (may be with new key-field data).

Converting the Get Next (GN) and Get Hold Next (GHN) Calls

Note: The DL/I call and command interfaces will be discussed together in this section of the manual. Remember that in the command interface there is no such thing as an SSA, but it is functionally replaced with a SEGMENT phrase. The equivalent to a qualified SSA is a SEGMENT phrase with a WHERE clause.

Also remember that all GN commands are internally translated to HOLD type calls and thus the programmer must be aware of the fact that DELETE and REPLACE commands can follow all GN commands.

The GN and GHN calls sequentially retrieve the next segment in a forward direction in the database, irrespective of any specification of one or more qualified or unqualified SSA. In a hierarchical database, sequential retrieval is always carried out from top to bottom, and from left to right, and considering the segments specified in the PCB used.

If no SSAs are specified, the next sequential sensitive segment (according to the PCB used) is retrieved. This is the segment following the position pointer at the completion of the previously successful call. Other SSA settings are described in the following:

- One unqualified SSA or SEGMENT

  The first occurrence of the specified segment type found by searching in a forward direction from the current position is retrieved.

- One or more qualified SSA or SEGMENT

  The path leading to the segment retrieved is defined by the SSA or SEGMENT content. The last specified SSA or SEGMENT defines the retrieved segment.

- When the last SSA or SEGMENT is qualified, it defines the unique segment that is to be retrieved. Higher-level qualifiers define the unique segments that are part of the path to the segment that is retrieved.

The GN call or command should be carefully converted. You should examine the related program logic in detail.

Conversions for the following combinations of GN calls or commands are described:

```
Single  GN   -  with    an SSA or SEGMENT qualified or unqualified
             -  without an SSA or SEGMENT

Multiple GNs -  with    SSAs or SEGMENTs qualified or unqualified
             -  without SSAs or SEGMENTs
```

Note: The 'next segment' is always determined by the previous DL/I call position. Therefore, when converting, you should know this position and then you are able to read the appropriate 'next' data in SQL/DS.

The 'next' data can be stored either in:

- The next row of the same table
- A (the next) row of another table.

Figure 4 on page 106 shows an arrangement of segment structures in DL/I and the corresponding tables in the SQL/DS environment. The figure is used to explain the conversions of the DL/I GN and GHN calls into SQL calls.

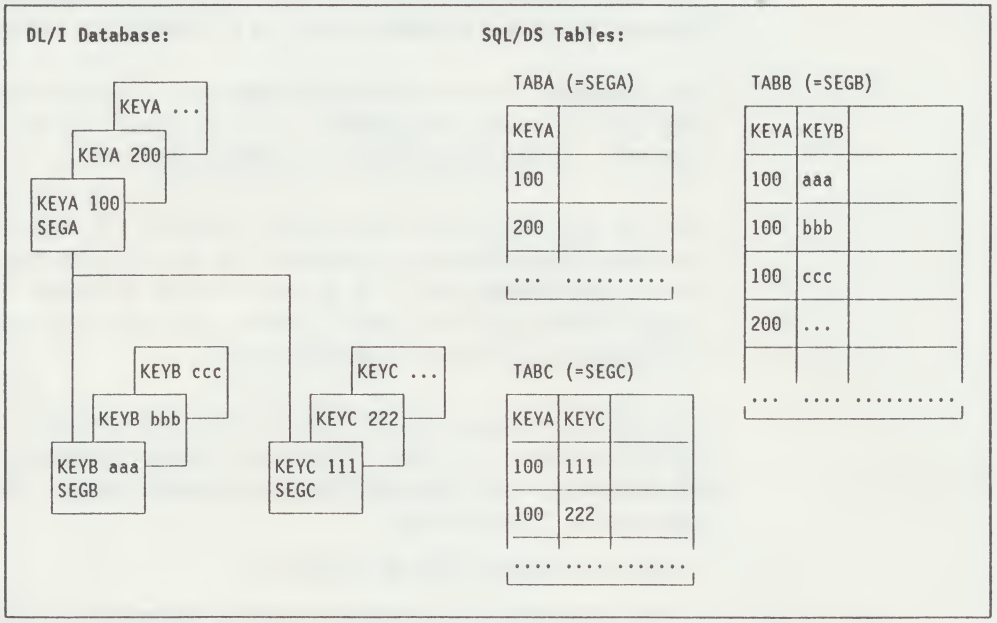


Figure 4. Example Structures Used in Describing the Conversion of DL/I to SQL.

A Single GN Call or Command with a Fully Qualified SSA or SEGMENT

The SSA or SEGMENT qualification corresponds to the next segment, which is derived from the current position. The last SSA or SEGMENT in the call indicates which segment must be read.

Using the example structure shown in Figure 4:

- It is assumed that the current position is SEGA KEY = 100, and the program initiates the call or command
- GN SEGA
- Then SEGA with KEY = 200 is read.

If the SSA or SEGMENT is qualified, and the SSA or SEGMENT operator is '= ', and the SSA or SEGMENT key is unique, then the GN call or command can be converted into a SELECT statement without the use of a cursor. (However, this is a rather rare case.) If the expected result is more than one row (key is not unique) then a cursor must be used as described for the following call or command.

A Single GN Call or Command with an Unqualified SSA or SEGMENT

Using the example structure shown in Figure 4 and assuming an unqualified SSA or SEGMENT, the result of a SELECT call or command would find several rows in the table (and SQL/DS would give you an error SQLCODE -810, equivalent to SQLSTATE 21000). Therefore, to convert individual GN calls or commands with unqualified SSA or SEGMENT operators properly, you must use a cursor.

You should always determine the current position (corresponding to DL/I). Is the GN intended to follow a twin chain, go to the next sibling, or jump levels in the hierarchy (change parent or child)? Knowing this enables you to read the 'next' data.

Given you are at the position SEGB with KEYB=ccc, (i.e. the last SEGB for KEYA=100) and you issue another unqualified GN call or command, you will get in DL/I the data of segment SEGC with KEYC=111. As it is usually not known in advance which key is contained in the segment C, you need to use a cursor. This is the case for all calls or commands of this type.

The DECLARE CURSOR statement is coded prior to references to the cursor in the program:

```
EXEC SQL
DECLARE cursorname CURSOR FOR
SELECT  colX,colY,...
FROM    TABC-VIEW                       Read from table via a VIEW
WHERE   KEYA-COLUMN = :HOSTVARA   <<<<< corresponds to KEYA=100
END-EXEC.                               in TABA or TABC
```

Note: If sequence of retrieval is important, then an ORDER BY statement must be included in the cursor definition.

The following should be coded where the DL/I call or command was:

```
MOVE SSAC-KEYFIELD TO HOSTVARA. Set the host variables used in the WHERE clause
                                of the DECLARE cursor statement.

EXEC SQL                    SQL/DS logically materializes all data as
OPEN cursorname             defined in the cursor (SQL/DS internal).
END-EXEC.

EXEC SQL                    Loop to retrieve all rows and store them
FETCH cursorname            in the host variables of the application program.
INTO  ..,...,...            See the explanation following.
END-EXEC.

EXEC SQL                    The cursor is no longer needed and is therefore
CLOSE cursorname            closed.
END-EXEC.
```

For the first FETCH in this cursor, you will get the data with KEYA=100 and KEYC=111 in table C. For each table required for such processing you need a separate cursor. You can use this order (OPEN, FETCH, CLOSE) as required. It allows you to get exactly the same data as you currently get in DL/I. (If you want the same data *in the same sequence* as you you currently get in DL/I you must add an ORDER BY clause to the DECLARE CURSOR statement.) However, in these cases, you should also check whether the processing sequence as provided by DL/I for a GN call or command is still required in SQL/DS. This may simplify your conversion efforts.

Sometimes, in DL/I, multiple PCBs are used to keep more than one position at a time. In this case, you should use and open multiple cursors. Remember, OPEN/CLOSE of a cursor loses position and incurs processing overhead, so minimize such activity for best performance.

Several GN Calls or Commands (Loop) with an SSA or SEGMENT

An SSA or SEGMENT can be qualified even if the GN call is part of a loop. This can be the case when either the SSA or SEGMENT key is not unique, or it is an operator other than '= '.

Independent of whether the SSA or SEGMENT is qualified or unqualified, you must use a cursor together with an EXEC SQL FETCH statement to read the same data as was provided with the GN loop.

The OPEN CURSOR is done prior to the beginning of the loop. The cursor definition (using the WHERE clause, and subsequently the ORDER BY clause), must be defined according to the previously defined DL/I call or command. The CLOSE CURSOR is done after and outside of the loop.

It is important that you check exactly the conditions under which the GN loop (to be replaced by EXEC SQL FETCH) ends. If this occurs when all the data of this type is read (end of data), then the corresponding SQL loop will end with the SQLCODE of +100. If the loop ends because of a certain value (field content) of a segment, then, as a rule, you can define these conditions usually in the DECLARE CURSOR statement (by a WHERE clause), and you can again get the SQLCODE of +100 at the end of a set of data.

If the loop is ended by specific application coding, for example, by using the contents of the Key Feedback Area in DL/I, then you must check the appropriate key fields, which are now host variables. Usually you can also do this by defining a WHERE clause with the cursor.

One or Several GNs (loop) without an SSA or SEGMENT

This situation may be more difficult to re-create. In the relational database, processing that corresponds to that in the hierarchical system needs access to several tables in a predetermined (similar to DL/I) sequence. To do this you usually need several cursors; for each segment type to be read, you should define and use a cursor for the corresponding table. Two things should be especially considered:

1. Check the PCB used in DL/I to find out which segments are processed. This will show you which tables you have to process.
2. Check if the application program requires a processing sequence of data. If not, this might ease your conversion effort.

The Get Hold Next (GHN) Call

The functional similarities between the GHN and the GN calls mean that you can migrate these calls in the same way. For GHN calls immediately followed by DLET or REPL calls, see the discussion of the GHU call on Page 98. Note that the associated SQL UPDATE or DELETE will sometimes be cursor-controlled and sometimes cannot be cursor-controlled (those cases where the cursor contains an ORDER BY clause).

Converting Get Next within Parent (GNP) and GHNP Calls or Commands

These calls or commands sequentially retrieve the next sensitive segment in a forward direction within an established parentage, with or without the specification of one or more qualified SSA or SEGMENT. The parentage of the segment must have been established by a successful GU, GHU, GN, or GHN call or command either immediately before this call or command, or at some prior time, provided no other call or command changes that parentage in the interim period. GNP or GHNP calls or commands cannot establish or change the parentage.

Here is an explanation of the SSAs or SEGMENTs:

- With no SSAs or SEGMENTSs specified, the next sequential sensitive segment within the established parent path is retrieved. The next segment is the one following the position pointer at the completion of the last successful call or command.
- If a single, unqualified SSA or SEGMENT is specified, the first occurrence of that segment type found in the forward search of the established parent is retrieved.
- If one or more SSA or SEGMENT is specified, the path leading to the retrieved segment is as defined by the multiple SSAs or SEGMENTs.

  The SSAs or SEGMENTs must be for segments at lower levels than the previously established parent. The last specified SSA or SEGMENT determines the segment retrieved.
- When unqualified SSAs or SEGMENTs are specified in a call or command with multiple SSAs or SEGMENTs, they establish the first occurrence of the associated segment type as a part of the path.
- Any missing SSAs or SEGMENTs between the previously established parentage and the segment type to be retrieved are generated as unqualified SSAs or SEGMENTs. This has the effect of establishing the first occurrence of the segment representing the missing SSA or SEGMENT as part of the path.

Similar to the GN or GHN calls or commands, you should know the current position. This means you must understand what DL/I is reading next. To do this you must know the 'parentage'. The segment read with the GU, GHU, GN, and GHN calls or commands is declared as the parent. The subsequent GNP or GHNP calls or commands read all segments below the parent segment. What segments are read depends on the SSA or SEGMENT qualification and on the definition of the PCB used during the DL/I calls or commands.

GNP Call or Command with an SSA or SEGMENT

A GNP call or command with an SSA or SEGMENT is handled in the same way as a GN call or command with an SSA or SEGMENT. If the SSA or SEGMENT is qualified with the operator '= ', and the SSA or SEGMENT key is unique, then the GNP call or command can be converted by means of a SELECT statement. If this is not the case, then a cursor must be used.

If the GNP call or command is part of a loop you must always use a cursor.

Using the example structure in Figure 4 on page 106, and looking at the following DL/I call or command sequence:

```
GU     SEGA   (KEYA  =  100)

GNP    SEGB   <----+  (the GNP call in a loop)
                    |
                    |
                    +
```

the conversion to SQL should look like:

```
EXEC SQL
DECLARE cursorname CURSOR FOR
SELECT ...,...,...
FROM TABB
WHERE KEYA = :HOSTVAR
END-EXEC.
```

Note: If sequence of retrieval is important, then an ORDER BY statement must be included in the cursor definition.

and

```
MOVE 100 TO HOSTVAR.

EXEC SQL                    immediately
OPEN cursorname             in front of the
END-EXEC.                   loop

EXEC SQL            <----+  Loop to get
FETCH cursorname         |  all rows
INTO ...,...,...         |
END-EXEC.                |
                         +

EXEC SQL                    immediately
CLOSE cursorname            after the
END-EXEC.                   loop
```

Note: The sequence in which the rows are retrieved may not be the same as in DL/I unless an ORDER BY is placed in the declare cursor.

When complete (all rows found) SQL/DS returns an SQLCODE of +100 corresponding to the DL/I status code GE.

GNP Call or Command without an SSA or SEGMENT (Unqualified GNP)

With this type of call or command, if only one segment type under the parent is read, it can be treated exactly as a GN call or command without an SSA or SEGMENT.

If there are several GNP calls or commands without SSAs or SEGMENTs, and there are several segment types under the parent to be read, you must, in this case, (as with GN calls or commands) use cursors, and allocate one cursor per segment (table) read.

If the program loop was ended by a status code other than a 'GE' (such as a 'GA' or 'GK'), then only specific tables must be read. This also must be considered in the corresponding SQL calls. In the same way you must consider the content of the PCB used. The PCB defines the segments (tables) processed.

The Get Hold Next within Parent (GHNP) Call or Command

Note: Remember that if the application program you are converting uses the DL/I Command interface all GNP commands are treated as GHNP calls and therefore the following discussion applies to all GNP comands.

The functional similarities between the GHNP and the GNP calls mean that you can migrate these calls in the same way. For GHNP calls immediately followed by DLET or REPL calls, see the discussion of the GHU call on Page 98. Note that the associated SQL UPDATE or DELETE will sometimes be cursor-controlled.

Path Call Considerations

Note: The following conversion method only works successfully if there is only one segment meeting the SSA qualification present on all of the hierarchical levels of a specific DL/I call (call path).

A DL/I path call is a single call or command that retrieves (or inserts) multiple segments in a hierarchical path using:

- **Using the DL/I call interface**

  a command code (D) and multiple SSAs.

  In this case, the PCB has the processing option P, for example, GP or GIRP. In the program, where the SSA is declared, the normal qualifying parenthesis '(' is preceded by an asterisk '*' followed by the command code 'D' ("D" meaning "path call").

  Here is an example; it uses two SSAs.

  Example:

```
SSA1:
     SEGA....*D(KEYA.....= 100)

SSA2:
     SEGB....
```

  (each period represents a blank).

  Note: The first segments that satisfy SSA(s) in which the D command code is present are moved into the IOAREA. The D command code is not necessary for the last SSA in the call, since the segment that satisfies the last level is always moved into the IOAREA.

  The DL/I call would be:

```
CALL 'CBLTDLI' USING func_gu,dbpcb,db_ioarea,ssa1,ssa2
```

  This call would retrieve both segments and place them together in the relevant IOAREA.

- **Using the DL/I command interface:**

  An INTO phrase is used with each SEGMENT for each hierarchical level to be retrieved.

  In this case, the PCB has the processing option P, for example, GP or GIRP.

  Here is an example which uses two SEGMENTs:

```
EXECUTE DLI
     GET UNIQUE USING PCB(1)
     SEGMENT(seg1-name) WHERE(field1-name = field1-reference)
     FIELDLENGTH(expression) SEGLENGTH(expression) INTO(ioarea1)
     SEGMENT(seg2-name) WHERE(field2-name = field2-reference)
     FIELDLENGTH(expression) SEGLENGTH(expression) INTO(ioarea2)
END-EXEC.
```

Conversion into SQL

The conversion into SQL, presuming that there is only one segment to be retrieved at each level, involves using the following:

```
MOVE 100 TO HOSTVAR.

EXEC SQL
SELECT ...,...,...            (specify only needed columns)
INTO   ...,...,...
FROM TABA I, TABB D           (joining the tables)
WHERE I.KEYA = :HOSTVAR (means key in TABA = host variable)
AND   I.KEYA = D.KEYA         (means key in TABA equals key in TABB)
END-SQL.
```

Note: This conversion method only works successfully if there is only one segment present on all of the hierarchical levels of a specific DL/I call (call path). If there are more, then a cursor must be used.

A Path Call or Command with Several Dependent Segments

In principle, this type of path call can be migrated in the same way as just described. However, you must check how the program processes the multiple segments read after the path call. This is especially important in view of the current position and the parentage aspects of the segments in case of subsequent GN, GHN, GNP or GHNP calls.

The migration to SQL would require the use of a cursor, as more than one row of data is retrieved.

The corresponding DECLARE CURSOR statement to replace this path call would require multiple predicates in the WHERE clause. The number of tables referenced, and the number of keys used in the WHERE clause correlate with the number of levels spanned by the DL/I GET call.

In the application program, prior to references to the cursor you must define the cursor.

For example:

```
EXEC SQL
DECLARE cursorname CURSOR FOR
SELECT ...,...,...
FROM TABA I, TABB D
WHERE I.KEYA = :HOSTVAR (means key in TABA = host variable)
AND   I.KEYA = D.KEYA         (means key in TABA equals key in TABB)
END-EXEC.
```

Note: If sequence of retrieval is important, then an ORDER BY statement must be included in the cursor definition.

Then retrieve the rows by opening the cursor, fetching the rows, and then closing the cursor.

For example:

```
MOVE 100 TO HOSTVAR.          correlates to key value in SSA

EXEC SQL                      immediately
OPEN cursorname               before the
END-EXEC.                     path call

EXEC SQL                  <-- Loop to retrieve
FETCH cursorname              all rows of data
INTO ...,...,...
END-EXEC.

(other appl. logic)       ---

EXEC SQL                      immediately
CLOSE cursorname              after the
END-EXEC.                     path call
```

Note: The sequence in which the rows are retrieved may not be the same as in DL/I unless an ORDER BY is placed in the declare cursor.

Applicability Of Outer Join

Note that when using a DL/I path call, if the lower level segment is not available, DL/I returns a 'GE' status code but it still places the higher level segment(s) in the I/O area. Although it is unusual, check to see if the application ignores the 'GE' and processes the high level segments even when the lower level segments are not present. If so, this is a more difficult situation to migrate to SQL/DS. This type of processing is logically equivalent to what is termed an *outer join* relational operation.

If you join two tables on column values, SQL/DS will only return those rows that match. So if you have declared a cursor with a join in its SELECT clause, when using (fetching from) that cursor, you will not see any rows from the first table that do not have a match in the second table to which the first is joined.

To achieve information retrieval equivalent to the path call described above, you want to see rows from the table corresponding to the parent segment even where no matching row exists in the table corresponding to the dependent segment. This is known as outer join (this example is actually a dissymmetric join) and while there is no single operator in SQL/DS SQL that performs this, it can be done in at least two ways:

- Your application could use two cursors, one for the matching rows, and one for those that do not match, to retrieve all occurrences of the rows of the table corresponding to the parent segment.
- The SELECT clause of your cursor declaration could retrieve a horizontally concatenated (joined) result by using a combination of the join, a UNION, a NOT IN predicate, and literals, such as:

```
EXEC SQL
DECLARE cursorname CURSOR FOR
SELECT A.PKEYA, A.COL1, A.COL2, B.COL1, B.COL2
  FROM TABA A, TABB B
 WHERE A.PKEYA = B.FKEYA    -- PKEYA and FKEYA are the joining columns
UNION
SELECT A.PKEYA, A.COL1, A.COL2, literal1, literal2
  FROM TABA A
 WHERE A.PKEYA NOT IN (SELECT TABB.FKEYA FROM TABB)
END-EXEC.
```

(Note that the literals in the above statement are 0 or blank(s) depending on the data type of B.COL1 and B.COL2, respectively[7].)

Other References

For details of Command Codes in SSAs, refer to *DL/I DOS/VS Application Programming Reference* (SH12-5411).

Migration of the command code L is discussed in 7.4.4, "Iterative (Set) Processing" on page 125.

Setting of Search Keys

In a hierarchical database system such as DL/I, access to a particular segment is achieved by going through the fixed levels in the database hierarchy. In the relational database system, you do not have to do this and can go directly to the table row(s) you want to process by specifying search condition(s) in the WHERE clause. You need only retrieve the rows required. To be able to specify the definition of relationships, as in DL/I, SQL/DS allows you to define relations between the tables. The relational database system uses the concept of foreign keys to define these relations. The differing techniques are illustrated with the aid of the following figure:

[7] To be precise, the columns and literals being UNIONed must match as to type and length. So if one of the columns is character, the literal should be SUBSTR('@@@...', 1, L) where @ is your fill character (e.g. blank) and L is the length of the character column. Similarly, you may have to SELECT, for example, INTEGER(B.COL1) if B.COL1 is SMALLINT, because the literal 0 is generated as an INTEGER, not a SMALLINT.

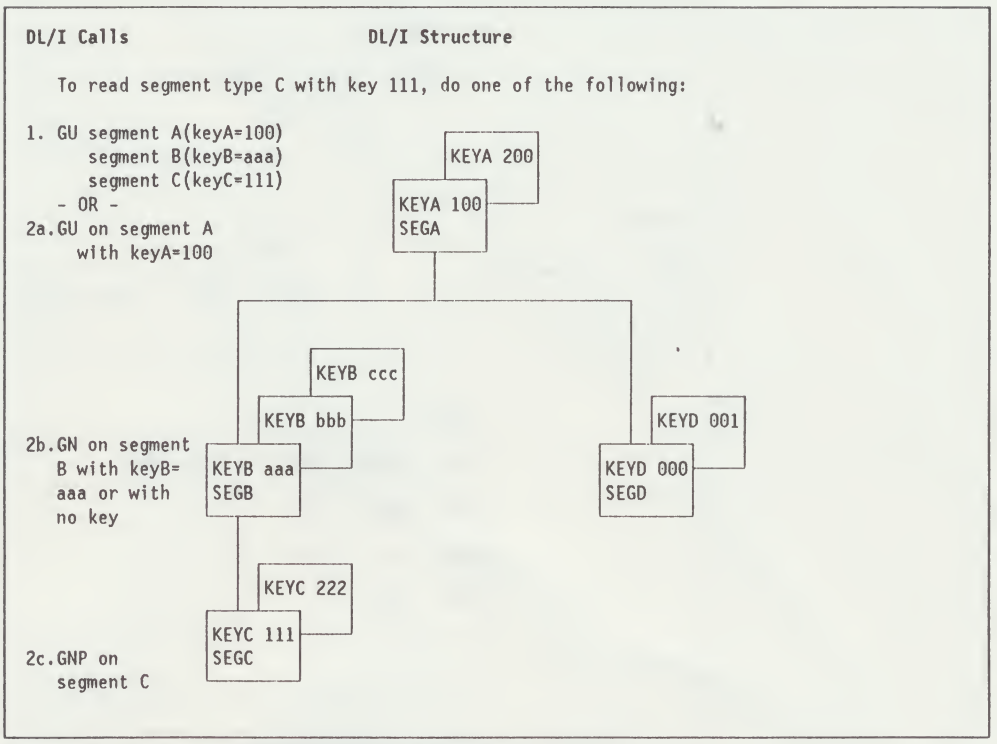


Figure 5. Positioning Calls in a DL/I Structure

To access segment C (SEGC) with the KEYC=111, DL/I must go through the hierarchical structure of the database to get there (that is, through segment types A, B, and C) using the calls shown in the figure.

In the relational database system you use the foreign keys KEYA=100 and KEYB=aaa and, in addition, KEYC=111 to gain access directly to table C.

In a relational database, segment C would be represented as a table as shown in the following figure:

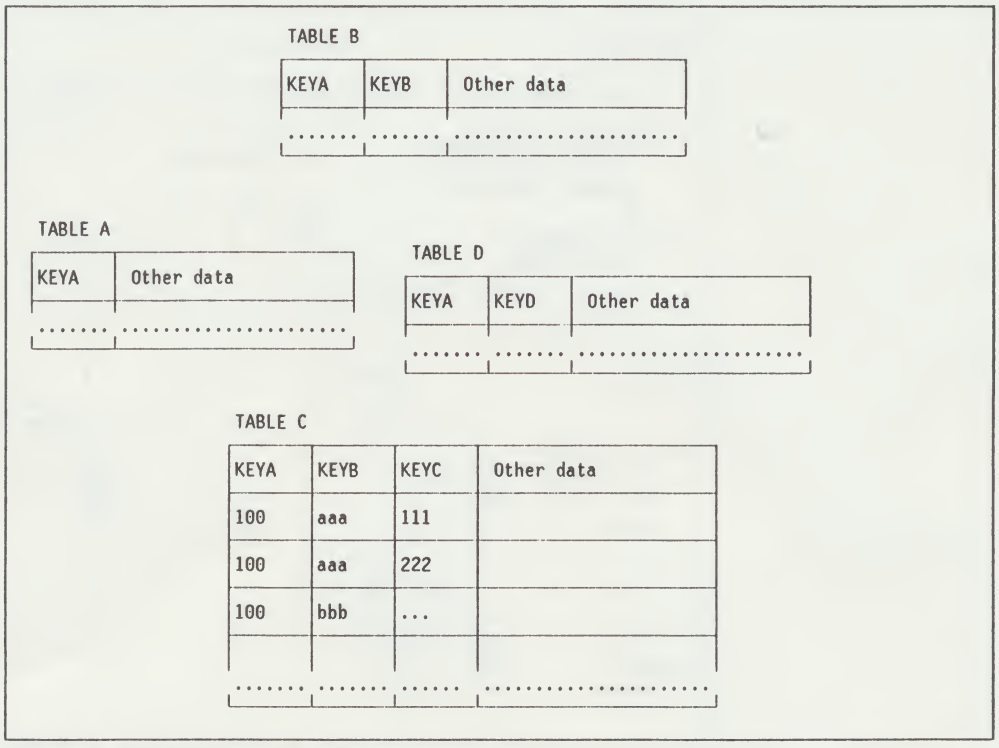


Figure 6. SQL/DS Table with Foreign Keys

To access the row with KEYC=111 in table C, you need the keys KEYA and KEYB. Usually, these keys are already stored in the appropriate SSA or SEGMENT key fields. You could either use these fields as host variables, or remove the SSA or SEGMENT key fields and move the key data into the appropriate host variables instead. There are usually the same number of moves as there are SSA or SEGMENT key fields.

If the SSA or SEGMENT key fields do not contain the key value, the DL/I application program reads with a GN call and then a GNP call that only defines the segment type.

You must use two cursors:

- With the first cursor, you read (open cursor and then FETCH) all data from table B (which contains the data of segment B) WHERE KEYA=100. Doing this you get the KEYB data.
- A second cursor (or just a SELECT) is used to process the data of table C. The application program now has KEYA and KEYB data as required.

If the data from the original GN call were not actually used, i.e. if the GN call was *only* for navigation and establishment of position, then a single cursor referencing only Table C, but qualified on *both* KEYA and KEYB, would be recommended.

You could also, for example, read all rows of table C that have KEYC=111. There could be several rows with this key when KEYA and KEYB are not specified.

For programs with complex logic, which have the same segment type with different key values, it is recommended that you make the SSA or SEGMENT key

field a host variable in the host-structure copybook. (See 7.5.1, "Host Variables" on page 134 for more information.) The existing MOVEs in the SSA or SEGMENT key field can stay and, therefore, you will always read the correct key value.

7.4.2 Data Modification Commands

Note: In this section the DL/I Call and DL/I Command interfaces are discussed together. The application program you are converting will use either the DL/I Call or DL/I Command interface. If the DL/I Call interface is used then SSAs are used; if the DL/I Command interface is used then SEGMENT and WHERE clauses will provide the same function as the SSA provides in the DL/I Call interface. In the DL/I Command interface the SEGMENT is qualified if it has a WHERE phrase. Therefore, as you read this section the term SSA can be interpreted as SEGMENT if your application was written using the DL/I Command interface.

Converting the Delete (DLET) Call or Command

The DLET call deletes from the database the last segment successfully retrieved by a GET HOLD (GHU, GHN, or GHNP) call, or GET command (GU, GN, GNP), and all of that segment's dependent segments or children at all levels beneath it.

No SSAs are required, but an unqualified SSA is allowed to select the segment to be deleted from a path of segments retrieved with a path call. In all other cases the unqualified SSA is ignored, and a qualified SSA gives a return code.

Unlike DL/I, the relational database system does not require a read before delete. This means that it is possible to remove the previous GET HOLD call if it is associated with the DLET call only.

DLET Call or Command on Single Segments

The following conversion technique can always be used when a single segment, of either a single or twin chain occurrence, is deleted.

If a single segment (qualified) was retrieved by a GHU or GHN call (or a GU or GN command), and only this segment is to be deleted, then you can convert this call in one of the following ways:

1. If the previous GHU or GHN call (or GU or GN command) can be removed, then the DLET call can be converted into the following SQL:

```
MOVE SSA-KEY-VALUE TO HOSTVAR.

EXEC SQL
DELETE FROM VIEWname (if using a VIEW of the table)
       WHERE KEY-COLUMN = :HOSTVAR
END-EXEC.
```

2. Similarly, the same SQL call can be used if the DLET call is preceded by a GHU or GHN call (or GU or GN command) that cannot be removed. The GHU or GHN call (or GU or GN command) is replaced by a single SQL SELECT (SELECT without cursor).

3. If the GHU or GHN call (or GU or GN command) cannot be removed, and the GHU or GHN call (or GU or GN command) has been converted to SQL

using cursors (non-unique key, for example) then the DLET can converted as follows:

```
EXEC SQL
DELETE FROM VIEWname
       WHERE CURRENT OF cursorname
END-EXEC.
```

Note: This may not be used if the cursor contained an ORDER BY clause.

DLET Call or Command on (Multiple) Dependent Segments Under One Parent

In DL/I such a delete operation is carried out with a "GHU or GHN - DLET" loop. If in the relational database system all these dependent segments are in the same table, they can be deleted by one SQL DELETE statement.

For example, using the structure shown in Figure 4 on page 106, to delete all the segments of type SEGB (under key = 100) in a loop, the DL/I code to do this would perform the following logic:

```
GU      SEGA   (KEYA  = 100)

┌─►
│  GHNP   SEGB
│  DLET
└──       Loop, to remove all twins
```

Changing this to SQL, for the corresponding tables shown in Figure 4 on page 106, you get:

```
MOVE SEGA-SSA-KEY-VALUE TO HOSTVAR.

EXEC SQL
DELETE FROM TABB
       WHERE KEYA = :HOSTVAR
END-EXEC.
```

This technique works well provided that the twins to be deleted are not to be read (e.g., for the purpose of a panel display). If the GET calls cannot be removed, and they have been converted to SQL using cursors, then the migration is done in the same way as described for the DLET call of single segments, using the WHERE CURRENT OF clause.

DLET Call or Command on a Parent with All Dependent Segments

In principle the conversion technique is the same as you would use if the delete was for a single segment. First you should check if the previous GET HOLD call (or GET command) can be removed; if so, you can delete the corresponding table rows.

The tables that belong together should be related by foreign keys in the table definitions. This is required to make use of the referential integrity functions of SQL/DS. The referential integrity functions of SQL/DS delete all related table entries through the CASCADE delete rule. Be aware that SQL/DS supports only one level of cascading delete. If the dependent segments deleted by a DLET call or command on a parent are on more than one hierarchical level, then mul-

tiple SQL/DS statements will be necessary. The implicit deletion of DL/I dependent segments will have to be done explicitly in SQL/DS.

Converting the Replace (REPL) Call or Command

The DL/I REPL call or command either replaces or updates the last segment successfully retrieved by a GET HOLD call type (GHU, GHN, or GHNP) or any GET command.

Note: Remember that in the DL/I command interface all GET type commands are translated internally to GET HOLD type calls.

The GET HOLD call must directly precede the REPL call and use the same PCB. No key fields change their value, only non-key data changes. No SSA is required with this type of call and if one is qualified an error may result (perhaps due to a mismatch with the segment currently retrieved by the GET HOLD call).

If the GET HOLD call (or GET command) can be removed and the rows to be updated can be identified with a WHERE clause, then GET HOLD and REPL can be replaced with a single SQL UPDATE statement:

```
MOVE SSA-KEYFIELDs TO HOSTVAR-keys.     (setup the access keys)

EXEC SQL
UPDATE VIEWname
   SET column-list = :HOSTVAR-list
WHERE KEY-COLUMNs = :HOSTVAR-keys
END-EXEC.
```

If the GET HOLD call (GHU, GHN, or GHNP), or GET command (GU, GN, GNP), cannot be removed (that is, it has to be migrated) then the clause FOR UPDATE OF must be included in the DECLARE CURSOR statement which replaces the GET HOLD call (or GET command). The following example allows for multiple REPL calls or commands to be handled within the looping logic of the application. In this situation the preceding GET HOLD call (or GET command) and the REPL call or command can be replaced as follows:

```
EXEC SQL
DECLARE cursorname CURSOR FOR
SELECT  ...,...,...
FROM    VIEWname
WHERE   KEY-COLUMN1 = :HOSTVAR1  (predicates for all columns
  AND   KEY-COLUMN2 = :HOSTVAR2   needed to identify rows)
FOR UPDATE OF column-list
END-EXEC.
```

Note: In the FOR UPDATE OF clause you should specify only the columns you plan to update (i.e. the "column-list").

The REPL call or command is converted into SQL as:

```
MOVE SSA1-KEYFIELD TO HOSTVAR1.  Set the host variables used in the WHERE clause
MOVE SSA2-KEYFIELD TO HOSTVAR2. of the DECLARE cursor statement to identify the
                                row to be updated.

EXEC SQL                    SQL/DS logically materializes all data as
OPEN cursorname             defined in the cursor (SQL/DS internal).
END-EXEC.

┌─►
│ EXEC SQL
│ FETCH cursorname                To read the data.
│ INTO  ..,...,...
│ END-EXEC.
│
│ MOVE data into HOSTVAR-list     Setup variables to update columns.
│
│ EXEC SQL
│ UPDATE VIEWname
│ SET column-list = :HOSTVAR-list  To update the data.
│ WHERE CURRENT OF cursorname
│ END-EXEC.
│
└─◄            Loop, to process all rows needing replacement
               This may involve changing the key values.
```

Note: The sequence in which the rows are retrieved may not be the same as in DL/I unless an ORDER BY is placed in the declare cursor. If an ORDER BY is used then this method of updating cannot be used.

Note: The list of columns (column-list) is positionally related to the list of host variables (HOSTVAR-list). This is also a good place to point out that even though we show a view being used in the cursor SELECT clause, that view must be for only one table. In SQL, you are not permitted to do an insert, update, or delete via a VIEW which joins tables.

Converting the Insert (ISRT) Call or Command

This call or command inserts a new segment into a database. It can also be used in a similar way to load segments into a new database, but loading of SQL/DS data should be done using SQL/DS utilities because of better performance. A specific 'LOAD' processing option such as in DL/I is not required in SQL/DS.

There are two types of ISRT call or command:

1. ISRT calls or commands for segments that have a unique key (coded as (...SEQ,U) in the corresponding DL/I database definition), or segments that have no key, but occur only once in the database.
2. ISRT calls or commands for segments that have 'multiple' keys (keys that may have duplicate values) (coded as (...SEQ,M) in the corresponding DL/I database definitions), or segments that have no key, but have multiple occurrences (per parent segment). These segments are inserted into the database according to the insert rule (FIRST, LAST or HERE) defined in the DL/I database definition (DBD).

You must decide which type of ISRT applies before you attempt to convert the DL/I call.

ISRT Calls or Commands on Segments with a Unique Key or Single Occurrence

When you convert this ISRT call or command to an SQL INSERT you have two possibilities:

1. If the DL/I ISRT call or command adds a root segment, the call can simply be replaced by an SQL INSERT for the corresponding table.
2. If the DL/I ISRT call or command adds a dependent segment, then you must consider that the concatenated key is part of the table row. This data must be moved into the corresponding host variables first. You should look at your table layouts to see the complete key of the row.

The following is an example of this type of conversion:

The DL/I INSERT is:

```
                                                qual qual unqual
CALL 'CBLTDLI' USING func_isrt,dbpcb,db_ioarea,ssa1,ssa2,ssa3
```

and the SQL INSERT statement is:

```
MOVE  SSA1-KEYFIELD  TO  HOSTVAR1.   It is assumed that the last part of the key
MOVE  SSA2-KEYFIELD  TO  HOSTVAR2.   (usually the target segment key) is
                                    already moved.  Move other non-key
                                    fields as required.

EXEC SQL
INSERT
INTO   VIEWname
       (KEY-COL1,KEY-COL2,COL3,...)
VALUES
       (:HOSTVAR1,:HOSTVAR2,:HOSTVAR3,...)
END-EXEC.
```

ISRT Call or command on Segments with Non-Unique Key and Insert Rule FIRST or LAST

In DL/I, FIRST and LAST insert rule causes subsequent retrieval to be, respectively, LIFO (last in, first out) or FIFO (first in, first out). In the relational database system, the only way to guarantee the sequence of retrieval is with the SQL ORDER BY statement. Therefore, if sequence of retrieval must be maintained, it will be necessary to use TIMESTAMP and/or DATE columns.

If you want retrieval to resemble that which would occur with a physical insert rule of FIRST in DL/I, you must specify DESCENDING in the ORDER BY clause on the TIMESTAMP and/or DATE column of a later SQL SELECT or cursor declaration. No specific INSERT consideration is necessary.

In case of insert rule LAST, you specify ASCENDING in the later SQL SELECT. No specific INSERT consideration is necessary.

An index on the timestamp and/or date columns should be used, based on the processing sequence desired. (Two indexes can be used to support frequent usage of descending *and* ascending processing.) This will improve performance when the FIRST inserted or LAST inserted "row" is the target of a select statement.

ISRT Call or Command with Insert Rule HERE

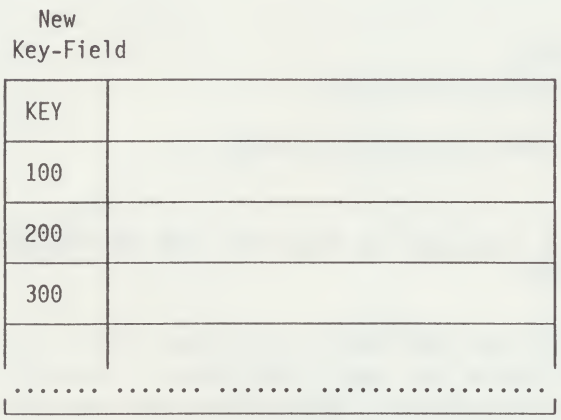
There are various possibilities to convert this DL/I call to SQL/DS. Two are described here to stimulate your thinking. Realize that you must understand the data and processing intent to choose the best conversion technique.

It is possible that the data has no predetermined processing sequence. In this case, simply insert the data utilizing the same key values used in DL/I without any additional columns added. This is the simplest case and relies on the power of the later SELECT statements to retrieve the data as needed.

If the sequence of the data is important (i.e. is based on the relative position of the segments) then you may consider the following technique.

In building the table which replaces the segment where the INSERT call with insert rule HERE is to be implemented, you create new numeric keys. Each time you insert a new row you increase this key by an incremental value. (The column could be defined as an integer or small integer data type.) The incremental value should be high enough to produce gaps that are large enough so that you can later add new rows in between as required and as shown in the following example. Volatility and randomness of the data are factors to consider when deciding upon the size of the incremental value.

Example:



In this example table, the gaps are large enough for up to 99 additional inserts. If more inserts are required, you should use a higher incremental value.

Note that you should thoroughly understand how the inserting application determines where 'HERE' is, if the retrieval sequence is important. Look for preceding GET calls for the same segment type, and subsequent analysis of the values therein.

A Path Call with an ISRT Call

If the DL/I ISRT call is made in association with a path call, you should consider to which segments the ISRT is made. Multiple SQL INSERT statements must be used, one for each table into which rows are inserted. The data for the SQL INSERTs must be moved to the relevant host variables. The INSERTs to the appropriate tables are handled the same as for other INSERT statements.

Note: Do not forget to prepare all key data in the corresponding host variables for each table you update.

DL/I Positioning vs. SQL/DS Cursor-Controlled Operations

DL/I positioning within a database has already been discussed in 7.4.1, "Navigation Commands" on page 95.

Set-Level UPDATE and DELETE vs. Record-At-A-Time

DL/I processing is oriented to do a single record at a time for updating and deleting. Multiple updates or deletes are achieved by iterative processing (i.e. looping).

SQL/DS allows the application to execute one statement to update or delete sets of data (without looping), if the update or delete is of a general nature (i.e. applies across a set of data or rows).

Due to this basic difference in the processing nature of the two database systems, thought should be given to eliminating the loop processing required in DL/I that would not be needed in SQL/DS. Examples have been given in conjunction with the respective DL/I REPL and DLET call sections previously.

7.4.3 Referential Integrity

Whenever a data element in one data structure (e.g. DL/I segment or SQL/DS table) depends on or refers to a data element in another data structure, there exists a need for the integrity of that reference to be maintained. There are various methods for this **referential integrity** to be maintained.

In DL/I, there are three ways of maintaining referential integrity:

- Structure of the database hierarchy

When a parent segment is deleted, all child segments within that hierarchy are deleted automatically. This structural integrity is maintained by the database management system based on relationships defined by the database administrator. Applications are coded to respond to the various status codes which DL/I returns to the application.

- Logical relationships

When a parent segment is deleted in one database, all logical children (whether in the same physical database or in the logically related database) may be deleted, depending on some rules. This structural integrity is maintained by the database management system based on relationships defined by the database administrator. Applications are coded to respond to the various status codes which DL/I returns to the application.

- Application programming (no use of DL/I features)

The programmer can enforce referential integrity within the application by coding to check for existence of a needed data element in one database or file before proceeding with the changes for the dependent data element. This method may be used in the SQL/DS environment also.

In SQL/DS, referential integrity is considered to be the automatic enforcement of referential constraints. A referential constraint defines a relationship between a parent table and a dependent table. A parent table contains a primary key, and a dependent table contains a foreign key. The referential con-

straint states every value of a foreign key in a dependent table must appear as a value of the primary key in the parent table, or be NULL. This ensures data consistency. The referential constraint is created when the foreign key is defined, and it is related to the primary key.

A referential constraint can be defined when a table is created, or added after the table has been created. This is unlike DL/I, which requires a DBDGEN or PSBGEN to accommodate changes in the processing options. When you define a referential constraint, you specify a primary key, foreign key, and delete rule for that relationship. Delete rules specify the action taken with respect to dependent rows when the parent row is deleted.

Key Changes (i.e. update of primary key)

Whenever the primary key is changed, there are programming considerations for referential integrity. The program must take appropriate action if an INSERT, UPDATE, or DELETE statement fails due to a violation of a referential constraint. Note that a column defined as a primary key cannot be NULL. There must also be a *primary index* (which must also be a *unique* index) on the column(s) comprising the primary key.

If you are updating a parent table, you cannot modify a primary key for which dependent rows exist. SQL/DS enforces these constraints. If such an update fails, all changes made by the SQL statement are rolled back. The application must check the SQLCODE and take appropriate action.

For a DL/I application, the programmer has to determine whether segments should be inserted, deleted, or replaced via the physical path only or the physical and logical paths. Those rules are specified in the physical DBD and the PROCOPT= parameter in the PCB. These rules must be examined in order to migrate to SQL/DS tables and maintain the functionally equivalent referential integrity.

You should understand the implications of the various insert, delete, and replace logical relationship rule options specified on the SEGM RULES= parameters in the DBD:

- Physical
- Logical
- Virtual
- Bidirectional virtual (for delete only)

and for each of the above

- First, Last, or Here

Refer to the *DL/I DOS/VS Database Administration Guide* (SH24-5011) for details regarding these various options and examples for each. There are also matrices detailing segment rule combinations and the possible results from such combinations. This information will not be repeated in this publication.

Refer to the *SQL/DS Application Programming* (VM: SH09-8086, VSE: SH09-8098) for further details regarding SQL/DS restrictions in the following sections:

- Inserting into Tables with Referential Integrity
- Updating Tables with Referential Constraints

- Deleting from Tables with Referential Constraints

Correspondence of Logical Relationship Delete Rules to SQL/DS "ON DELETE" Clause

See "Referential Integrity" on page 67 in Chapter 5, "Database Design" for a discussion of similarities and differences of DL/I logical delete rules and SQL/DS delete options for referential integrity.

7.4.4 Iterative (Set) Processing

Some specific cases are described under:

"Special DL/I Processing Options"

"DL/I Command Codes"

"SSA Modifications" on page 126.

Special DL/I Processing Options

Some DL/I PCBs use the processing option GO (Get Only) and allows you to retrieve data that may have been only just updated and may not be committed yet. This is sometimes referred to as a "dirty read" or read without integrity.

This option is most often used in case of text data or for display only purposes, where exact currency of data is not critical, or viewing of uncommitted data is allowable.

In the SQL/DS relational database system, you cannot read uncommitted data (data that is currently being retrieved for update). Therefore, the GO processing option cannot be equivalently migrated.

However, the programs that use the PCBs with processing option GO could be replaced by the use of 'row locking' in the access module (or package) for retrieving the corresponding SQL/DS tables. This will allow maximum possible access to the data (while one row may be locked, and inaccessible, the other rows are accessible).

For more details about SQL/DS locks and lock modes, refer to *SQL/DS Application Programming* (VM: SH09-8086, VSE: SH09-8098).

DL/I Command Codes

Command codes are an optional addition to the SSAs that provide specification of the functional variations applicable to the DL/I call function. As discussed in "Path Call Considerations" on page 111, in SSAs the normally qualifying '(' (open parenthesis) or the blank if unqualified, can be preceded by an asterisk '*' followed by one or more command codes.

Some of the command codes can be converted through cursor operations, while others require changes to the program logic.

A relatively easy command code to convert is the Command Code L. This command code is used in an SSA to retrieve the *last* occurrence of a segment type (in a twin chain) under an established parent. If, for example, VIEWA corresponds to a DL/I parent 'SEGA', and VIEWB to SEGA's dependent 'SEGB', then the command code could be replaced by the SQL column function **MAX** shown in the following example:

```
EXEC SQL
SELECT  ...,...,...
INTO    ...,...,...
FROM    VIEW_B
WHERE   VIEW_B.KEY_A = 100  AND
        VIEW_B.KEY_B =
        (SELECT   MAX(KEY_B)
         FROM     VIEW_B)
END-EXEC.
```

For details of use of command codes in SSAs, refer to *DL/I DOS/VS Application Programming Reference* (SH12-5411).

SSA Modifications

SSAs[8] can be modified in a DL/I application program, which uses the DL/I Call interface, in several different places. You must look at the Cross Reference List of the COBOL compiler to check if one of the fields in the SSA is modified. If this happens, the DL/I call may look different at the time of execution than it appears from the source code.

If the program gets too complex as a result of many SSA modifications, you should make the modified SSA fields part of the copybooks that contain the host variables (see 7.5.1, "Host Variables" on page 134 for more details). The program can then use these fields and their contents like a host variable to control the converted SQL call.

Modifying the Operators[9]

Operators can be changed by the application program. For example, an initial value of '= ' in the operator field is changed in the application program to '> ='.

If the operator '> =' is used, the result may be more than one row and a single SQL SELECT call may lead to an error. An SQLCODE of -810 (equivalent to SQLSTATE 21000) indicates more than one row qualifies for the WHERE clause. Therefore, a cursor will have to be used. The cursor should have a WHERE clause with the '> =' operator. You need as many cursors as you have different operators (which are not '= '). This is because the operator in the WHERE clause of a cursor is processed only when the cursor is opened.

Modifications of the SSA Qualification[10]

It is possible to set the 9th column (position) of the SSA to 'blank', or the other way around (non blank), via the application program. This makes a qualified SSA unqualified or the other way around.

---

[8] If the DL/I application program being converted to SQL/DS uses the DL/I Command interface, it is unlikely that any of the considerations discussed here are pertinent. There are no SSAs in such a program. The equivalent SEGMENT phrases cannot easily be changed because of the syntax of the commands. Other than data contained in parameters marked reference it is not possible to modify SEGMENT parameters.

[9] It is not possible to dynamically modify the Operators when using the DL/I Command interface.

[10] It is not possible to dynamically modify the Qualification when using the DL/I Command interface.

Such a change to the SSA must be consistent in context with the DL/I call to be executed. An unqualified SSA (9th column is blank) has no influence on a REPL, DLET, or ISRT call. GN or GU calls may be influenced very much; for details refer to "Converting the Get Next (GN) and Get Hold Next (GHN) Calls" on page 105 and "Converting the Get Unique (GU) and Get Hold Unique (GHU) Calls" on page 95, respectively.

Modifications to the Segment Name in an SSA[11]

If the segment name specified in an SSA is changed it means that you have to access another table (assuming segment data were not combined into a single table). For example, if a modification means that segment X is accessed instead of segment Y, you must read the table that contains the data of the X segment instead of the table that contains Y data.

Any time a segment name is changed (hence table name change) you need another SQL statement.

If a cursor is required, you need an additional cursor that you have to open and close.

Modification of the Key-Field Name[12]

If the key-field name is changed by the application program you can treat this in the same way as a change of segment name. However, you are most likely still processing the same table. For details, see the previous description of "Modifications to the Segment Name in an SSA".

Modification of the Key Field Value[13]

If the key field content is changed by the application program you should also change the content of the host variables you are using in the WHERE clause of your cursor or in the SQL SELECT statement.

If you are using a cursor you must also close the cursor (if it is open) and open the cursor again after you have set the host variables in the WHERE clause of the cursor to the new values. Recognize that opening and closing a cursor has performance implications and should be minimized if possible.

If you still need the already open cursor because you must not lose your current position (for example, during a bill of material explosion) you need an additional cursor similar to the already open cursor.

---

[11] It is not possible to dynamically modify the Segment Name when using the DL/I Command interface.

[12] It is not possible to dynamically modify the Key-Field Name when using the DL/I Command interface.

[13] It is possible to dynamically modify the Key Field Value when using the DL/I Command interface.

7.4.5 Handling Special Cases

This part of the chapter describes how some specific cases may be handled in relational programs.

Date-Field Handling

Many application programs process date data. In the relational database system you have a DATE data type that is new to DL/I or COBOL programmers. This means it is recommended that you adapt your programs and copybooks to make use of the DATE data type provided by the relational database system. (By doing this you will also minimize 'year 2000' (turn of century) problems.)

It is recommended you use date-handling routines with functions such as:

Interface and work fields for date conversions

Providing current date from SQL/DS

Converting DL/I packed date to SQL/DS date

Converting SQL/DS date to packed date

Converting various external date formats to SQL/DS date format

Converting SQL/DS date to various external (output) date formats.

The common routines which perform date conversion (from external format to program internal format or the other way around), can handle the DATE format used in the relational database system as well. For dates in SQL/DS tables, the forming of the 9's complement is no longer necessary. It can be replaced by an ORDER BY ASCENDING or DESCENDING clause in the SQL statements (in the SELECT or in the cursor declaration).

The DL/I programs get the date from CICS for the online programs, and from the operating system for batch programs. This should be changed as the current date is also provided by the relational database system. SQL/DS supports various forms of date representation.

In the relational database tables, dates are defined with a data type of DATE, and have a length of 10 characters. In the host variables of the application program this means that a MOVE to or from the date fields, or date comparisons (in the application programs) of the date format, no longer match. In many cases, this is a conversion problem.

One method for dealing with this problem, which reduces the required changes to a minimum, is as follows:

- Use new names for the date fields in the host variables.
- Keep the DL/I date fields as in your relevant (program) copybooks (old name, old definition), but change from `PIC S9(6) COMP-3` to `PIC S9(7) COMP-3`.
- After reading a date, convert the contents of the relational database date field (host variable) into the DL/I date format using a date-handling routine. This allows you to keep the current application program unchanged (as regarding the date field).
- Before each INSERT and UPDATE, convert the contents of the old date-format field into the relational database date fields (host variable) using a date-handling routine.

This method also make it possible to minimize the impact that the new date handling has on your application programs.

The following example illustrates what has been described:

SQL/DS date field:

```
01 SPADDDTSQL     PIC X(10).   Host variable for date field
```

Current DL/I date field:

```
01 SPADDDT        PIC S9(7) COMP-3.
```

If the current DL/I date field is PIC S9(6) COMP-3, you should change it to PIC S9(7) COMP-3 so that it is suitable for the year 2000 support.

```
PERFORM FETCH/SELECT                    Read fields of a row that include a
                                        date field.

                                        Prepare interface to date-conversion
                                        routine.
MOVE SPADDDTSQL TO WS-DBIE-CHAR10.      Convert date from
PERFORM LABEL1 THRU LABEL1-EXIT.        internal (SQL/DS) to external.
MOVE WS-DBIE-PACK04 TO SPADDDT.         Move converted date in packed format
                                        into a work field.

IF SPADDDT GREATER SAVE-DATE            Actual
   MOVE SPADDDT TO SAVE-DATE            processing.
END-IF.
          ...

MOVE SPADDDT TO WS-DBEI-PACK04.         Converting again
PERFORM LABEL2 THRU LABEL2-EXIT.        external (panel or program)
MOVE WS-DBEI-CHAR10 TO SPADDDTSQL.      to internal (SQL/DS).


PERFORM INSERT/UPDATE                   Modification
```

At the end of the program:

```
LABEL1.
     EXEC SQL INCLUDE INTTOEXT END-EXEC.   Convert date from SQL/DS date format
LABEL1-EXIT.                               to date that is displayed or printed,
                                           or used internally by the
                                           application program.
LABEL2.
     EXEC SQL INCLUDE EXTTOINT END-EXEC.
LABEL2-EXIT.
```

Additional comments to the example:

1. After each FETCH and SELECT, where the date field is read, the contents of the host variable (SPADDDTSQL) is moved into a variable (interface to the date-conversion routine).
2. Then the routine (copybook) INTTOEXT is called.

   This would perform the conversion from internal (10-byte CHAR) format to external (4-byte PACKED) format (and other formats at the same time).

3. The converted date is moved to the field (SPADDDT), which is used in the application program's logic for processing.

For an INSERT and UPDATE call you must carry out this process the other way around. Instead of the routine INTTOEXT being called, the routine EXTTOINT is called, which performs the conversion from external to internal (SQL/DS) format.

If you use the date field as a key field (which is necessary for reading) the key field must be converted to internal format before it is used for reading the SQL/DS table.

Use Of Fillers

Fillers are generally used when you want to move the contents of date fields from one area to another where the field lengths differ or where extra space was reserved in the segment for unforeseen future use.

Two examples of this are given in the following.

Moving from a Host Variable to a Save Area Structure:: A program may move the contents of the host variables (I/O area) to a save area. This area may contain date fields that are declared as 4-byte packed fields. The host variable is a date field that is 10 bytes. Therefore the host variable and the save area structure do not match.

To carry out a MOVE from a host variable to a field of a different length in a structure you can pad out the length of the date fields with fillers in the save area to be 10-bytes long. Since SQL/DS does not support structures as host variables, each field in the save area must be filled separately.

The following example shows you how to do this:

Host Variables:

```
      .
01 SPADDDTSQL        PIC X(10).          <-- SQL/DS date host variable
      .

01 SPADDDT           PIC S9(7) COMP-3.   <-- DL/I date field
                                             (4 bytes)
```

Save Area:

```
01 SAVE-PRODST.
   05    . . .
   .     . . .
   05 SAVE-SPADDDT     PIC S9(7) COMP-3.
   05 FILLER           PIC X(6).
   05    . . .
   .
```

In the program:

```
MOVE SPADDDTSQL TO WS-DATE-EXTERN.

MOVE host variables to SAVE-PRODST.        Move all fields into the save area - the
⋮
MOVE SPADDDT TO SAVE-SPADDDT.        date field is wrong.
                                     It is corrected with the next statement.

PERFORM EXT-DATE-TO-INT-DATE THRU END-DATE-CONTROL   Convert SQL/DS date format
                                                     into a packed format by
                                                     the use of the
                                                     date-conversion routine.

MOVE PACK-DATE TO SPADDDT.
```

Moving from a Save Area Structure to a Host Variable:: In the reverse case, where the save area is moved to the host variables, the date field must also be moved separately from the save area into the DL/I date field. When this is done, the date must be converted from the external (packed) format into the SQL/DS date format.

The following example shows you how this is done (it uses the same host variables and save-area definition as in the previous example).

In the program:

```
MOVE SAVE-PRODST fields  TO host variables.
⋮
MOVE SAVE-SPADDDT TO WS-DATE-PACKED.
PERFORM INT-DATE-TO-EXT-DATE THRU END-DATE-CONTROL
MOVE WS-DATE-SEPARATED TO SPADDDTSQL
MOVE SAVE-SPADDDT TO SPADDDT.
```

Note: Since SQL/DS does not support structures as host variables, moves must be done one field at a time.

7.4.6 Column Selection vs. Segment Selection

There are various differences between DL/I and SQL/DS based on SQL/DS's use of columns versus DL/I's use of segments. SQL operations are performed on columns within tables. DL/I can operate on fields within segments, provided the processing options have been set up to accommodate this.

When migrating a DL/I application to SQL/DS, consider the following:

- What are the considerations for undefined portions of segment?
- What are the field-level sensitivity considerations?
- What application dependency exists on the concatenated key?
- What differences exist in data area/host variable declarations?

7.4.7 Error Handling

In the DL/I application programs the status code of DL/I is normally checked after each call. Depending on the status code, messages are provided, an ABEND is forced, or specific program logic is executed. All of this must be done in SQL/DS also. What is different is the status codes that are provided by SQL/DS.

The relational database equivalent to a DL/I status code is the SQLCODE.

The following chart contains the most frequently encountered DL/I status codes on the left-hand side and their SQL/DS equivalents on the right-hand side. Note that some special DL/I status codes have no corresponding SQLCODE. Refer to the DL/I and SQL/DS messages and codes publications for further detail regarding the DL/I status codes and SQLCODEs.

DL/I Status Codes	SQL/DS SQLCODEs
Blank	0
Successful completion	
GE	+ 100
Set not found	
No further segment exists (loop)	End of the data in the cursor set (WHERE conditions in the cursor correspond to the DL/I loop.)
GB End of the database (indicates end of set of data being processed)	No equivalent SQLCODE. + 100 shows the end of the data in the cursor set. (WHERE conditions in the cursor correspond to the DL/I loop.)
GK Segment change on the same hierarchical level ("switch to sibling") Probable end of logical set being processed.	No equivalent SQLCODE. Tables are read using cursors. + 100 shows each time the end of the data of a cursor set is reached.
GA Change upwards in the hierarchical level "change of parentage" Probable end of logical set of data being processed.	No equivalent SQLCODE. Tables are read using cursors. + 100 shows each time the end of the data of a cursor set is reached.
II	- 803
Segment/row with the same key already exists	

Figure 7. Comparison of DL/I Status Codes and SQLCODEs (partial list)

There are also SQL/DS-specific SQLCODEs. These codes usually fall into the category that is handled as a severe error by the application program. Severe error code processing may be put into subroutines or copybooks.

Online and batch programs probably need separate severe-error routines. You could make standard severe-error routines and provide them for use across the system. For example, you may choose to create:

Online SQL severe-error-handling routine

Batch SQL severe-error-handling routine

Online message-handling program

Batch message-handling program.

SQLCODEs that are considered severe errors cause a program break. For these types of errors an appropriate severe-error routine should be invoked to handle the situation.

This error routine could rescan the SQLCODE and operate as follows:

- If the SQLCODE signals a successful completion of the SQL call, the program could continue processing.
- In any other case, an abridged error message should be displayed at the workstation or a message provided at a hard-copy device, and the program ends with a rollback.
- In case of OPEN or CLOSE of a cursor, any SQLCODE other than 0 (zero) is considered a severe error.

Use Of EXEC SQL WHENEVER Statement

The return code indicated in the SQLCA indicates the success or failure of the previous statement's execution. If SQL/DS encounters an error while processing the statement, the return code in SQLCODE is a negative number.

Because the SQLCA is a valuable problem-diagnosis tool, it is a good idea to include in your application programs the instructions necessary to display some of the information contained in the SQLCA. Refer to the SQL/DS Application Programming Guide for further details regarding what should be displayed for different circumstances.

The WHENEVER statement allows you to make a decision in your program based on a condition that is indicated in the SQLCA. The WHENEVER statement looks like:

```
EXEC SQL
  WHENEVER condition action
END-EXEC.
```

Refer to the SQL/DS publications for details regarding the three possible conditions to specify:

1. SQLWARNING
2. SQLERROR
3. NOT FOUND

and the action you want taken

4. CONTINUE
5. GO TO label

7.4.8 Scrolling

Since DL/I applications rely on positioning within the hierarchy of the database, and maintenance of that position by DL/I, it is straightforward for the application to allow for scrolling forward. Backward scrolling is not as straight forward. Since DL/I does not have a 'GET PREVIOUS' call, scrolling backward across the retrieved data requires a DL/I positioning call to back up in the hierarchy and then retrieve the segments again. Alternatively, the application may store the retrieved segments and use CICS scrolling facilities for the backward scroll.

With SQL/DS there are no facilities to establish 'position' at the *end* of a set of result rows or to 'fetch backward' through a result set. That leaves your program with two options:

1. Keep a copy of the data that has been fetched and scroll through it by some programming technique.
2. Use SQL to retrieve the data again, typically by a second SELECT statement. For example, you could have two cursors open on the same table, with different ORDER BY clauses.

There may be two indexes on the sequencing column(s) used for scrolling, one in ascending sequence and the other descending. This will facilitate retrieval by SQL SELECT with ORDER BY in either direction and thus provide data in either sequence. These could improve the performance of the retrieval with two cursors described above.

7.5 COBOL Programs

This part of the chapter describes how some specific cases may be handled for COBOL programs.

7.5.1 Host Variables

For every relational database table there are corresponding host variables in the application program. They form the I/O area for the application program and correspond to the DL/I I/O areas. The major difference is however that in DL/I this I/O area must have the same layout as the physical segment of the database. In SQL/DS this need not be the case since SQL/DS operates on each column (use of SELECT * is not recommended). For the host variables of one table there should be a copybook.

The names of the host variables may have the same names as the field names in the previous DL/I I/O areas. The host variable name may differ from the column name in the SQL/DS table (but is not recommended).

The host variable definitions must correspond to the data types defined in the relational database table definitions.

The only Level Numbers supported by SQL/DS are 01 Level, 77 Level, and, in the case of varying length data types, 49 Level. This means that the structures typically used by DL/I for the I/O area must be changed as shown below.

All redefinitions within the structures must be dropped. A program is also easier to read when a variable is referred to by one name only.

Sometimes you still might need (want) a redefinition of a field. You might want to do this because it would mean fewer changes in the application programs during conversion. This can still be accomplished by moving the host variables to a save area structure and then using the redefine capability.

The example that follows illustrates the different implementation possibilities.

```
DL/I:                                    SQL/DS:

01 IOAREA.
   02 FIELD1.                              01 FIELD2     PIC X(3).
      03 FIELD2   PIC X(3).                01 FIELD3     PIC X(5).
      03 FIELD3   PIC X(5).                01 FIELD4     PIC S9(3) COMP-3.
   02 FIELD4      PIC S9(3) COMP-3.        01 FIELD5     PIC S9(9) COMP.
   02 FIELD5      PIC S9(9) COMP.

                                         01 SAVE-AREA.
                                            02 FIELD1.
                                               03 FIELD2   PIC X(3).
                                               03 FIELD3   PIC X(5).
                                            02 FIELD4      PIC S9(3) COMP-3.
                                            02 FIELD5      PIC S9(9) COMP.
                                         01 IOAREA1 REDEFINES SAVE-AREA.
                                             02 FIELD1    PIC X(8).
                                             02 FILLER    PIC X(6).
```

Figure 8. Comparison of DL/I and SQL/DS IOAREA's

Field1, as it was in DL/I, no longer exists in SQL/DS. However, converting the DL/I program to SQL/DS may be simpler if it did.This can be accomplished in the following manner:

- Define a SAVE-AREA structure as shown above.
- Redefine the SAVE_AREA structure with another structure (IOAREA1) as shown above.
- Move the host variables to the corresponding fields in the SAVE-AREA structure.

In the IOAREA1 structure Field2 and Field3 are overlayed by Field1, which can now be used just as it was in the DL/I program.

The reverse is also possible. For example, Field1 could be in the SQL/DS SAVE-AREA structure and Field2 and Field3 in IOAREA1. As a rule you should have the smallest field units (Field2, Field3) in the relational tables.

As is described in "Setting of Search Keys" on page 114 it makes sense to keep the SSA key fields as working fields. Where the same SSA may be used in several programs, the key field can be part of the copybook with the related host variables. The key field is defined in the copybook as a single variable definition.

In some copybooks you can find separate definitions of date fields. For some columns (host variables) there are also indicator variables declared in the relevant copybook. For more information refer to 7.5.2, "Indicator Variables" on page 136 and to "Moving from a Host Variable to a Save Area Structure:" on page 130.

The definitions for host variables, old SSA key fields, and indicator variables are required when using a table in an application program. Therefore, these definitions may be combined in a single copybook per table. Then if there are changes in the table, you have to change only the one copybook.

7.5.2 Indicator Variables

In the relational database system you must specify for each column, in addition to the data types, NULL (by *omitting* NOT NULL) or NOT NULL as described in the following:

NULL	Non-NULL value may be present, or the NULL value may be used (or defaulted to). Useful if contents may be unknown, etc. An indicator variable must be specified.
NOT NULL	At INSERT or UPDATE a valid non-NULL value must be provided.

If a column allows NULL, you should always use an indicator variable for processing the column. The indicator variable shows if the field contains the NULL value. This indicator must be set by the application program when a column or row is updated or inserted. The indicator is set by SQL/DS when the nullable column is read. Date fields and timestamp fields are defined as NULL when they are to be filled with data later, that is, not at the time when a row is created. SQL/DS does not allow invalid data in columns (for example, 0 in a date). See *SQL/DS Application Programming* (VM: SH09-8086, VSE: SH09-8098) for details and examples of use of host variables and null-indicator variables in conjunction with nullable columns.

7.5.3 Initialization of Fields (Defaults)

DL/I application programs often have special copybooks with predefined contents that are moved into the segment I/O area. Using this method, the segment I/O area is initialized with default values before any input data for an insert of a segment is moved. These copybooks may usually be removed in the course of migration to the relational database. The initializing of columns (fields) is achieved by MOVEs of single constants to the host variables, which are used during INSERT.

You may want to write your application so the columns are also filled with *your* default values before INSERT.

The MOVE statement of the default values for the host variables could be kept in one copybook (one for each table). The copybook would move the default values in the program before the INSERT statement.

Note: You should first set the initial (default) values for the host variables before you move actual input values (for example, from the panel).

7.5.4 Field-Level Programming

As a rule in DL/I you read a complete segment and the I/O area must have the same layout as the segment. In the relational database system, you select single columns and the data can be read into single host variables.

For the purposes of migration it is recommended that you create a VIEW of all the columns in each table. The view is used to process the table data.

Here are some examples:

Definition of the host variables:

```
01 FIELD1    PIC X(3).
01 FIELD2    PIC S9(4) COMP.
01 FIELD3    PIC X(10).
01 FIELD4    PIC S9(7) COMP-3.
```

The SQL for reading a complete row might be:

```
1.   EXEC SQL
          SELECT *
          INTO   :FIELD1,                  <------ Read all columns
                 :FIELD2,                         into single host
                 :FIELD3,                         variables.
                 :FIELD4
          FROM   VIEWname
          WHERE  COLUMNname = :FIELDname
     END-EXEC.
```

Note: In this case the list of fields in the INTO clause must be in the same order as the columns are defined in the CREATE statement used to define the view.

```
2.   EXEC SQL
          SELECT COLUMN2, COLUMN3, COLUMN1, COLUMN4
          INTO   :FIELD2,                  <------ Read selected
                 :FIELD3,                         columns into single
                 :FIELD1,                         host variables.
                 :FIELD4
          FROM   VIEWname
          WHERE  COLUMNname = :FIELDname
     END-EXEC.
```

Note: In this case the list of fields in the INTO clause must be in the same order as the columns are listed in the SELECT clause.

Figure 9. Sample SQL SELECT Statements.

The SQL coding shown in example 2 in the previous conversion example is perhaps the best. It allows you to specify in your SQL call the columns to be actually processed in a specific application program. This minimizes the impact of changes to the SQL/DS table (new or modified columns, or a new layout of a row).

In case of update, you should especially look at the key fields and whether you also want to modify the columns used to build an index. It is often better (for performance reasons) when you INSERT a row with the new key fields and then DELETE the row with the old key.

The following is an example of how to process (select) specific columns only:

```
EXEC SQL SELECT COLUMN2,COLUMN3
         INTO   :FIELD2,
                :FIELD3
         FROM   VIEWname
         WHERE  COLUMNname = :FIELDname
END-EXEC.
```

7.6 Unit Testing

All migrated programs must be unit tested to verify correctness. During a conversion, the results of the current DL/I program serve as a control for testing the converted version. If a test plan exists for the DL/I program, it can be used as a model for the SQL/DS program.

Data used in parallel tests must be consistent. Obtaining the data can be as simple as keying in the same transactions for online functions, or it can be as complex as writing special extract programs to get the data from DL/I.

Testing of Cross System Product applications is an integral part of the CSP/AD development cycle. For more information on testing these programs, refer to the Cross System Product manuals.

Testing batch programs also involves changing the JCL needed to run the program(s). For specific guidelines on changing JCL, see 11.4, "Job Control and Processing" on page 169 in Chapter 11.

7.7 Program Migration Checklist

This checklist can serve as a reference guide for the programmer during migration.

Documentation

1. Does the source program documentation need to be changed?
2. Do you have a copy of the DL/I structure diagram relating to this program?
3. Do you have a copy of the DL/I segment layouts?
4. Do you have a copy of the DL/I to SQL/DS data element cross reference?
5. Do you have a copy of the SQL/DS views and indexes for the tables to be used?

For batch programs, add:

6. Do you have a copy of the production JCL in which this program is executed?
7. Do you have run sheets for jobs that execute this program?

For online programs, add:

8. Do you have a copy of the map(s) for the DL/I program? BMS?
9. Do you have a copy of the map(s) for the Cross System Product application?

Coding

1. Have you added the INCLUDEs for SQLCA and views or tables used in the program?
2. Have you deleted the DL/I statements no longer needed from the program? (It is recommended that DL/I statements be commented out rather than actually deleted, and that adequate comments be added to document the rationale for the changes.)

3. Have you qualified the use of any existing names that are the same as new table columns?
4. Have you changed all occurrences of changed table and structure names?
5. Have you replaced all the DL/I calls or commands with the equivalent SQL statements?
6. Have you changed comments in the program to refer to SQL/DS operations, not DL/I operations?
7. Have you added SQL COMMIT statements as required?
8. Have you checked that column names use underscores instead of hyphens?
9. Have you replaced DL/I error handling routines with appropriate SQL/DS error handling routines?

Code inspection

1. Do you have an original copy of the source with deleted lines marked?
2. Do you have a new copy of the source with modified and inserted lines marked?

Testing

1. Have you obtained production data or a subset of production data and run the DL/I production version of the program?
2. Have you run the same data with the SQL/DS version of the program?
3. Have you used ISQL or QMF to compare modified tables with an on line query for the equivalent DL/I segments?
4. If a report or list is sorted, is the order correct?

For batch programs, add:

5. Have you compared any reports from the two programs?
6. If a control statement is required, have you tested invalid and missing control information?

For online programs, add:

7. If the application program does scrolling, have all possible situations been tested?
   a. Paging forward?
   b. Paging backward?
   c. What happens when less than a full screen of data is available?

Chapter 8. System Testing

This chapter details testing methods and procedures, emphasizing system integration testing. Batch, online, ad-hoc query, and cutover to production testing issues are also addressed. It is assumed that applications are moved into production in the SQL/DS environment while the DL/I system continues to run.

Testing may be one third to one half of the entire conversion project time and effort. A comprehensive test plan is important. In this test plan, measurable tests, acceptance criteria, test exit criteria, schedule dependencies and task relationships need to be planned and agreed to. The test plan also needs to cover fallback and recovery synchronization options.

8.1 Testing Methodology

Testing should be done on as stable an environment as possible. That is, minimize version changes to the operating system, compilers, database products, and other programs which may cause the results to differ from the original application system.

Even though the current applications seem correct, inconsistencies may be introduced by changes in the environment. For example, a COBOL program may compile different object code when using a new level of compiler. The resulting new executable code may produce different results. Thus debugging may include recompiling and executing the original application to isolate the source of inconsistencies.

Review any changes to the application made during the conversion to see if new test conditions are needed. All design or function changes should have associated documentation. This documentation will assist in test definition.

Testing is where you receive the benefit from a "No Design Change" philosophy (not making changes to the application except those related to the DBMS). The more consistent the inputs and environments are between the original and migrated systems, the less effort is required in test.

8.2 Performance

Performance of the new system should be considered at all stages, and measurements made of transaction and batch performance as soon as practical during migration. This will ensure that any areas of concern can be identified early and remedial action taken.

Once migrated, an application may take more or less CPU to run than before. Where built-in functions, set processing, and other capabilities of the relational DBMS can be used instead of navigational routines, SQL/DS will do more work with fewer lines of code. Where SQL/DS is forced to mimic functions that would not normally be part of an SQL/DS application design, CPU consumption may increase.

System testing should assume a performance perspective, in addition to checking for equivalent function. Testing must include some means of checking

the new environment (applications and DBMS), both in total for capacity planning, and individually so that any single transaction that appears to consume too many resources can be analyzed.

8.3 Testing Steps

System testing should be a combination of testing for equivalent function and for performance that meets the requirements. Figure 10 on page 143 shows an outline of the steps to be taken.

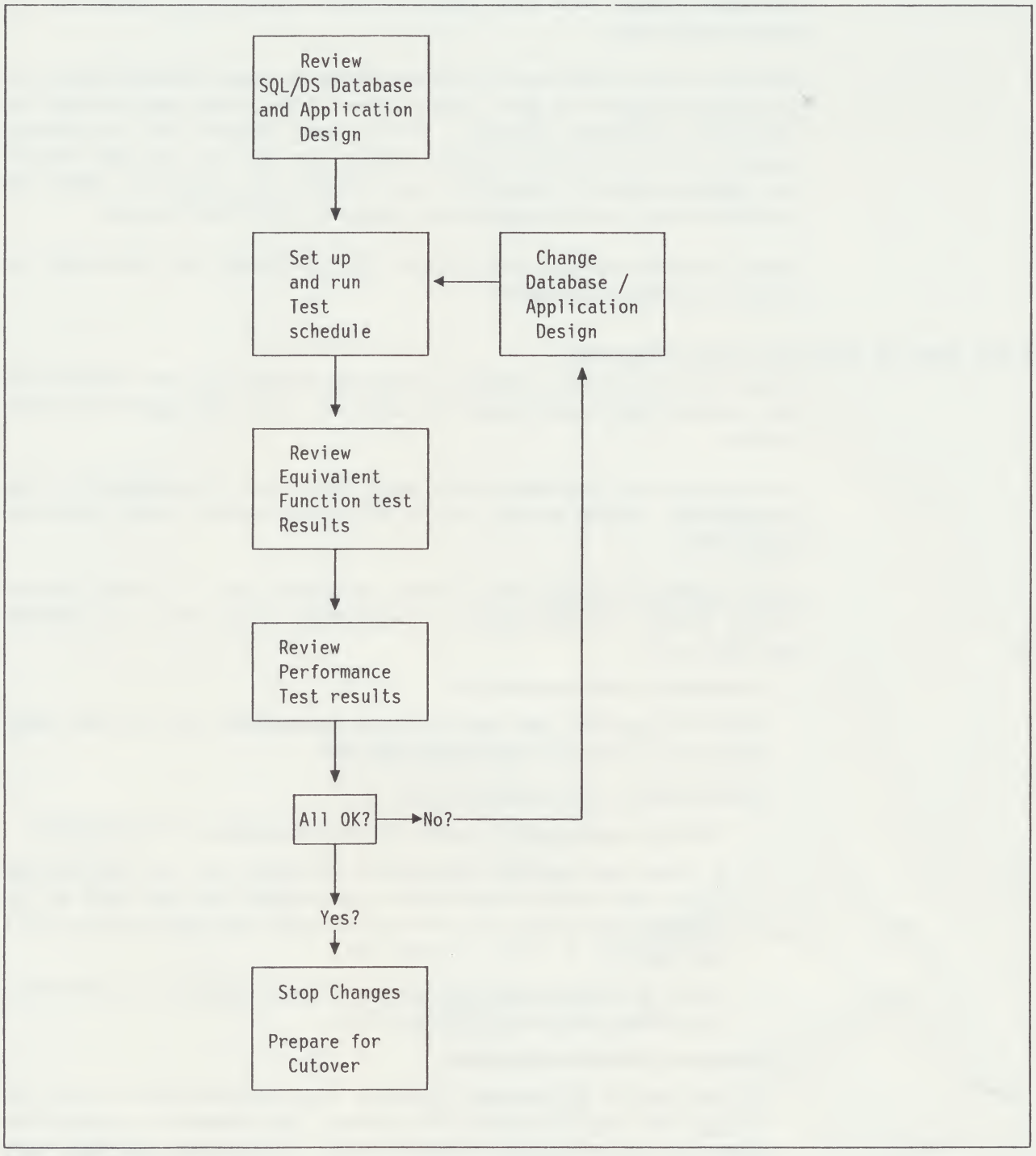


Figure 10. Testing steps

The following text explains the rationale behind each box in Figure 10.

8.3.1 Review SQL/DS Database Design

The database design should be reviewed as a whole prior to system testing. A formal review is useful so the project team ensures that all is ready for system test and becomes aware of any potential problem areas prior to system testing.

SQL/DS EXPLAIN provides information about the access paths available and which path was chosen by the optimizer for the SQL call. It is used to ensure

that where indexes have been defined to speed access to the data, they are actually being used.

EXPLAIN is used in development, but sometimes the small volumes of data in a development system can give a false picture of the access path that will be selected for production execution. Some catalog statistics can be manually updated, and it is vital that if system testing does not use data with volumes and makeup similar to production, these statistics are updated to reflect the production values before packages are prepared and EXPLAIN is used.

Refer to *SQL/DS Database Administration* (VM: GH09-8083, VSE: GH09-8095) for general performance guidelines.

8.3.2 Set up and Run Test Schedule

Formal testing should be managed step by step through unit test, function test, and complete system test in much the same way as any new application development.

The test effort may be divided several ways. Batch and online applications may be separated. Ad-hoc queries may be included in addition to the predefined applications.

Before starting, determine which testing method(s) to use. For more comprehensive testing, consider using a combination of the first two methods described below.

1. ***Transaction result comparisons***

   With this approach, the same series of transactions are run twice, once against DL/I data, once against SQL/DS data.

   Considerations of this approach are:

   - Testing is performed on results, allowing a test of equivalent function.
   - A testing tool like WITT (Workstation Interactive Test Tool) can be used to run both sets of transactions and compare the data fields on the screens, highlighting those that are not equal. For more details of WITT see Appendix B, "WITT" on page 179.
   - There is no guarantee that data that is not accessed or modified by a transaction stream is in fact equivalent.

2. ***Database contents comparisons***

   As described in the previous approach, a set of transactions can be run through both the DL/I and SQL/DS systems. The updated DL/I data is then copied to a second group of SQL/DS tables. Comparisons are then made between the two groups of SQL/DS tables.

   Considerations of this approach are:

   - If necessary, every database record can be compared.
   - Testing may highlight changes to 'related' data which might not be picked up using screen transactions.
   - A disadvantage is the amount of data to be compared. One application error could cause many fields to be in error, and there would thus be many messages to examine.

3. ***New Transactions***

   Here, a new system is created in parallel with the old system under conversion. That is, a new series of transactions is created, using copies of the original BMS maps. Standards are used to identify the names of the new transactions, which are then run against the copy of the data in SQL/DS. Since transaction codes are usually hidden in BMS maps, this approach can be almost transparent to end users.

   Considerations of this approach are:

   - The old and new are separate. If required, a period of parallel production execution, where transactions are entered into both systems, is possible.
   - It provides a way to clean up many libraries which may contain redundant copies of maps, programs, etc. The easy thing to do is concatenate new libraries and eventually scrap the old ones.
   - Extra effort is required to create the new environment.

To summarize, testing may be done as follows:

1. Select a testing methodology. Is it the same for batch and online?
2. Define and/or identify tests that establish that migrated application results are consistent with the original applications.
3. Document the test plan and the criteria to be used for acceptance of the converted application.
4. Create tests according to the plan. This includes tests which verify that cutover is complete through full-scale production.
5. Create a test DL/I database. Initially, this can be a small subset of the current production database. The value of a small amount of data is to minimize the computing costs and reduce time during first level tests. For full system testing this could be increased to as large a subset as feasible.
6. Stabilize the test DL/I database. Allow no updates during the unload. This will be used as the base for updates.
7. Create procedures for unloading the DL/I test database and loading SQL/DS test data in a logical order.
8. Create a test SQL/DS database which represents data equivalent to the test DL/I database.

   In VM a separate SQL/DS database machine can be defined for the test database. In the native VSE environment, multiple user mode supports a single SQL/DS partition. Hence, the test "data base" would be a set of test tables defined in an existing database.
9. Load the SQL/DS tables with data migrated from DL/I.
10. Create a test scenario for DL/I applications.
11. Repeat the test using the same keystrokes for the SQL/DS applications.

8.3.3 Review Equivalent Function Test Results

All three testing methods require that the output of each application be scanned for equivalence before and after conversion. Output may be in the form of screens, reports and external files. Identify and reconcile the differences. Where the differences are not acceptable, change the application and retest. Several tools can be used to test equivalence.

In the VM environment, use the CMS LISTFILE command to ensure that the record formats are the same, that the number of records are consistent, and that the logical record length is identical. Then use the CMS COMPARE command to examine the two CMS disk files (fixed or variable length format) on a record for record basis and to display dissimilar records at the terminal. Optionally, specific columns can be compared. VM spool facilities can be used to obtain a hardcopy of the compare results.

When scanning files, with these commands or a pattern matching program, masks over certain character strings or locations which are not necessary to compare for equivalence may be used. Time of day may be reported in the page header, for example. It will change from one run to the next, and would be flagged as different if not masked as not necessary to compare.

A program such as WITT is designed for regression testing, and can store and compare copies of screens, report discrepancies, and recognize and skip occurrences of such items as timestamps during the compare, thus reducing the number of differences reported. With all of these capabilities, WITT is one recommended approach.

QMF or ISQL is appropriate for ad-hoc queries, and can supplement other planned testing. If the second testing method (database contents comparison) is chosen, QMF or ISQL can be used to compare data in the SQL/DS database, and the SQL/DS database created directly from the DL/I database. Allowing users to interrogate the data will accomplish testing and education at the same time.

Compare Application Results

1. Run both sets of transactions through, save the results, and compare the screens.
2. Highlight and check any discrepancies.
3. Create further scripts to interrogate and compare the changed ('after') DL/I contents with the changed SQL/DS contents.

Compare Database Contents

1. Unload the updated DL/I database and load the data into a new set of SQL/DS tables.
2. Compare the SQL/DS tables as updated by the migrated applications, against the SQL/DS tables loaded from the updated DL/I data in the previous step, using the results of ad-hoc queries and query applications. The order SQL/DS actually stores the data on the physical medium (DASD) is not necessarily the same sequence of bits, so you cannot compare the actual database contents bit by bit in the order SQL/DS stored it.
3. Alternatively, unload both DL/I data and SQL/DS data into sequential files and compare using a utility or program.

With either method, review any discrepancies and, if required change the SQL/DS database or application(s) and repeat the process.

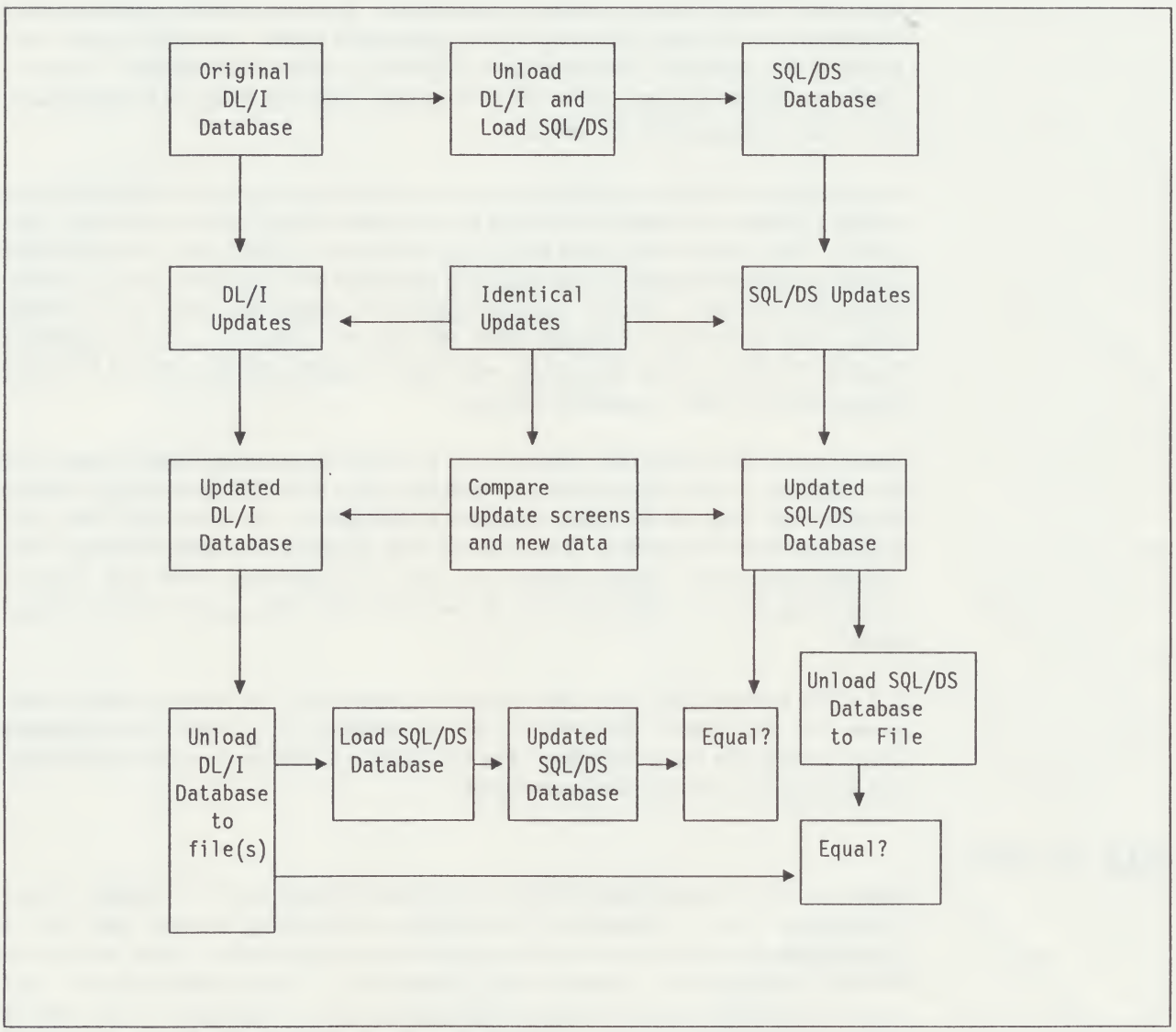


Figure 11. Process To Compare Database Contents

8.3.4 Reviewing Performance Tests

To effectively anticipate the load and performance of the new system, measurements should be made during the test runs. First, measurements should be collected using the old system for future comparison purposes; similar measurements can then be taken for the new system. Be careful here to compare the correct figures as CPU and elapsed time measurement methods may be different when using SQL/DS.

Even if not at production transaction rates, the monitors can measure the CPU resource needed to run single transactions and the increased resource required for production loads. Similarly for batch programs, the figures obtained can be scaled up to allow for the number of records to be processed during a production batch run.

Elapsed time is more difficult as it depends on many factors. However, provided the system used for testing is getting reasonable service from the host operating system, the elapsed times obtained should be representative of the times that should be obtainable in production. Be careful not to misinterpret a measurement indicating that service is apparently better. For most online transactions, the majority of elapsed time in taken up with I/O operations. If, due to relatively small volumes, data stays in buffers, elapsed times may increase in production compared to the tests.

A good way to monitor performance tests is to set up a spread sheet; CPU and elapsed times can then be entered for individual transactions and batch programs. The spread sheet can be set up to increase the figures for each transaction for the anticipated load and then calculate the total load, with a similar approach for batch. It can also be used to highlight the high load problem areas and any wide variance from the figures obtained before migration. Further, if any changes are made, new figures need only be entered for those transactions or batch programs affected.

When transactions are run, SQL/DS can produce accounting records which can be matched to the transactions. However, the SQL/DS accounting records provide only part of the data required to determine the total CPU time consumed by that transaction. You will need to combine the data collected from various monitoring tools (unique to each environment) with the SQL/DS accounting data, in order to arrive at the total CPU time consumed by a transaction.

For CICS transactions, the CMF data can provide the necessary data to complete the calculation. For batch, VSE accounting can provide the necessary data to complete the calculation. For CMS, VM accounting can provide the necessary data to complete the calculation.

8.3.5 All OK?

Note that any comparison technique will show differences in the results. Some differences may be expected. For example, time-of-day stamps may not be appropriate to compare for equivalence, though the relative value may be of interest to determine elapsed time performance. Also, comparing the data before and after conversion may show differences in ordering in the case of unsequenced data.

All differences should be investigated and efforts made to determine the cause, but it should be realized that it is unlikely that every field will show the exactly the same results first time around. You may need several iterations of changing and checking, before results are considered good. As this process happens, there will be diminishing returns for the effort expended and some thought should be given to what the critical data are before the process starts. Some failing comparisons may not be worth pursuing and some functional changes may not affect the integrity of the data or the way that users process that data.

For example, where data is listed in certain order on a screen, this may or may not really be important to a user.

A similar perspective may apply to performance questions. It is unlikely that every transaction will perform the same as before. Some will take more

resources and some fewer. The work prior to this can be used to determine which transactions are critical, and minimum acceptable performance levels.

At this point, elapsed times and resource utilization of standard utilities may also be evaluated. Estimates should have been made during the database design, and any critical times can be reviewed and tested if necessary.

8.3.6 Change Database / Application Design

We recommend you first open a change log to keep track of changes and effects.

Next, be selective. Resist any tendencies to change several things at once or it will be very hard to determine cause and effect. Where changes overlap, make only one change that affects a given transaction or module at a time. This will be the quickest route to completion in the long run.

The reason that performance is evaluated at the same time as function is that both application and database design affect performance. If code and an SQL statement must be changed, it is good practice to ask if performance was good to start with, and to keep performance criteria in mind as changes are decided.

Where there is a performance concern for certain transactions or modules, there are two classes of changes that can be made:

1. Physical changes - those that do not affect the application
2. Logical changes - those that do affect the SQL and thus the programs.

Examples of physical changes are:

- Addition or removal of an index
- Changes to the columns in a given index
- Ordering the data in a table a different way
- Combining small code tables into one table and giving applications views which make the data appear as many small tables.

Examples of logical changes are:

- Merging two tables into one (denormalizing)
- Splitting tables horizontally or vertically
- Setting up separate history tables
- Setting up 'sparse index' tables
- Setting up a 'bucket' column table

Physical changes should always be considered first. They only require changes to the database and a PREPARE of the application package to ensure the new structure is used.

Logical changes mean that every application involved with that table(s) will need to be examined and possibly changed. (However, the impact of some types of logical changes - such as denormalization - may be reduced by the use of views as described in 5.4.2, "Designing for Relational" on page 61 in Chapter 5, "Database Design.")

8.3.7 Preparing for Cutover to Production

Production cutover includes moving migrated applications from test to production, and loading SQL/DS tables with data from the appropriate DL/I database. This differs from bringing a new application online since much data must be unloaded and reloaded into SQL/DS. Whereas the program libraries can be switched fairly quickly, the unload and reload may take some time, and the affected data will be unavailable for update during this time.

For small and pilot changes, it may be possible to set up a parallel run where, even in production, operators enter each transaction twice; for applications where large amounts of data are involved, this may be difficult or impossible because of the amount of DASD required or workload required.

Cutover procedures should include a fallback plan. Once in production, a fallback plan should allow for cutover to be tested, yet retain the ability to go back to the cutover point. It is important to test trial runs of the cutover. The plan should also include some point at which the migration is considered 'operational'. After this point, any problems will be fixed rather than attempting to fall back.

A sequence of events preparing for cutover may include the following:

1. CREATE SQL/DS tables.
2. ALTER any columns as necessary if you are adding to an existing SQL/DS database. Populate any ALTERed columns as appropriate.
3. Stop all updates to DL/I data.
4. Copy existing DL/I data to cartridge/tape.
5. Copy existing BMS to cartridge/tape.
6. Copy existing applications to cartridge/tape.

   (These last three will allow for swift fallback if needed.)
7. Obtain DASD space needed during the cutover (e.g. for load files, work data sets, etc. This may be a good time to back up and/or delete unneeded files.)
8. Unload the data from DL/I that is needed for loading into SQL/DS.
9. Load the SQL/DS tables. (This should ensure that the data conversion programs handle all conditions existing in production data.)
10. UPDATE STATISTICS for DBSPACEs and related indexes, creating statistics that will be used to determine the optimum access path to the data.
11. PREPARE all application programs or packages. Access path selection is optimized based on the current data so it is important to do the PREPARE after the SQL/DS tables are loaded with production data, and statistics have been updated.
12. Ready all migrated applications and related files for production. Switch libraries if appropriate, or copy programs into production libraries. Bring new transactions online and disable old ones.
13. Verify database contents using your method. All testing of converted applications is complete at this point.

14. Update run sheets, flowcharts, and related operational documentation for production.

Applications which do not change the contents of the database may continue to run with the DL/I production data until the time the equivalent application is running in production using SQL/DS.

Online System Verification

This is a basic verification that the migrated system is accessible and the screens are functional, including queries and updates.

1. Test user access to the migrated production application system. This involves actually logging on to the system.
2. Invoke each screen in the migrated system.
3. Run the tests selected to verify that the application system meets requirements. This may be considered an acceptance test.

Online verification may accomplished by running WITT as suggested before. This allows a playback or replay of the CICS terminal activity. Online verification may be tested once on the original system. Then, to ensure that results are consistent between the original and migrated systems, the terminal interaction may be replayed in the exactly the same sequence as necessary to produce the desired consistency on the new system.

Note that you should not perform updates now since you are using production data. Any changes made as a test will need to be exactly reversed before production execution.

Batch System Verification

Compare all reports and control totals between the original and migrated systems. Similar to the online verification test, the batch applications may also have a test set defined which uses online queries to compare results after a batch run.

8.4 Production Execution

Once all verification is complete, migrated applications can be put into production. **Before this:**

1. Take a database archive of the newly loaded SQL/DS system.
2. Disable the old transactions or application system.
3. Ask the users to delay updates to the new system for (e.g.) an hour to ensure the main functions are working.
4. Ask the users to manually record critical updates to the new system for (e.g.) a day so that they can reenter them in case of a problem.
5. Archive migration libraries.
6. Archive cutover datasets.
7. Remove the DL/I areas from backup procedures after the data is no longer required.

Chapter 9. DL/I Security Migration

Security facilities to prevent unauthorized access to data are provided by both DL/I and SQL/DS; however, the implementation of these facilities is completely different in these two products. The purpose of this chapter is to give the reader an understanding of these differences and thus an appreciation of what must be done relative to security in order to migrate data and programs from DL/I to SQL/DS.

The first and most significant difference is in the basic security philosophy of the two products. DL/I allows access to data unless the user is specifically restricted. SQL/DS takes exactly the opposite philosophy and denies all access unless the user is specifically authorized to perform the requested function.

Both products use the facilities of the environment in which they are running to prevent unauthorized access to transactions and programs. Thus the existing transaction manager security facilities of CICS can continue to be used after migration of DL/I data to SQL/DS. If the migration is to VM then the security features of VM can be used with SQL/DS.

The security facilities in SQL/DS are much more extensive than those in DL/I. The following sections will give an overview of these facilities.

The difference in philosophy and a few other differences in the way that SQL/DS security operates mean that a minimal security plan must always be put together for SQL/DS to ensure that security needs continue to be met, even if DL/I security has not been used extensively.

9.1 DL/I Security

Security may be applied to DL/I data at different levels:

1. CICS can control which USERID is allowed to run a given transaction and program. This may be accomplished by using the CICS internal security system.
2. The DL/I PSB (Program Specification Block) is referenced by the transaction and may contain one or more PCBs (Program Communication Blocks). The PCB will specify which segments of a DL/I database can be accessed and which types of functions a program may perform on those segments. Programs may be allowed to retrieve, update, delete, insert, etc., according to the PROCOPTs (processing options) specified on the PCB. The default PROCOPT is to allow all functions.
3. Where security is needed down to field level within a segment, field level sensitivity may be coded within the PCB.
4. Where data is especially sensitive, segment edit/compression routines may be coded. These exits work at segment level and may encode, edit or compress data before storage in the DL/I database.

Figure 12 on page 154 shows an outline of DL/I security.

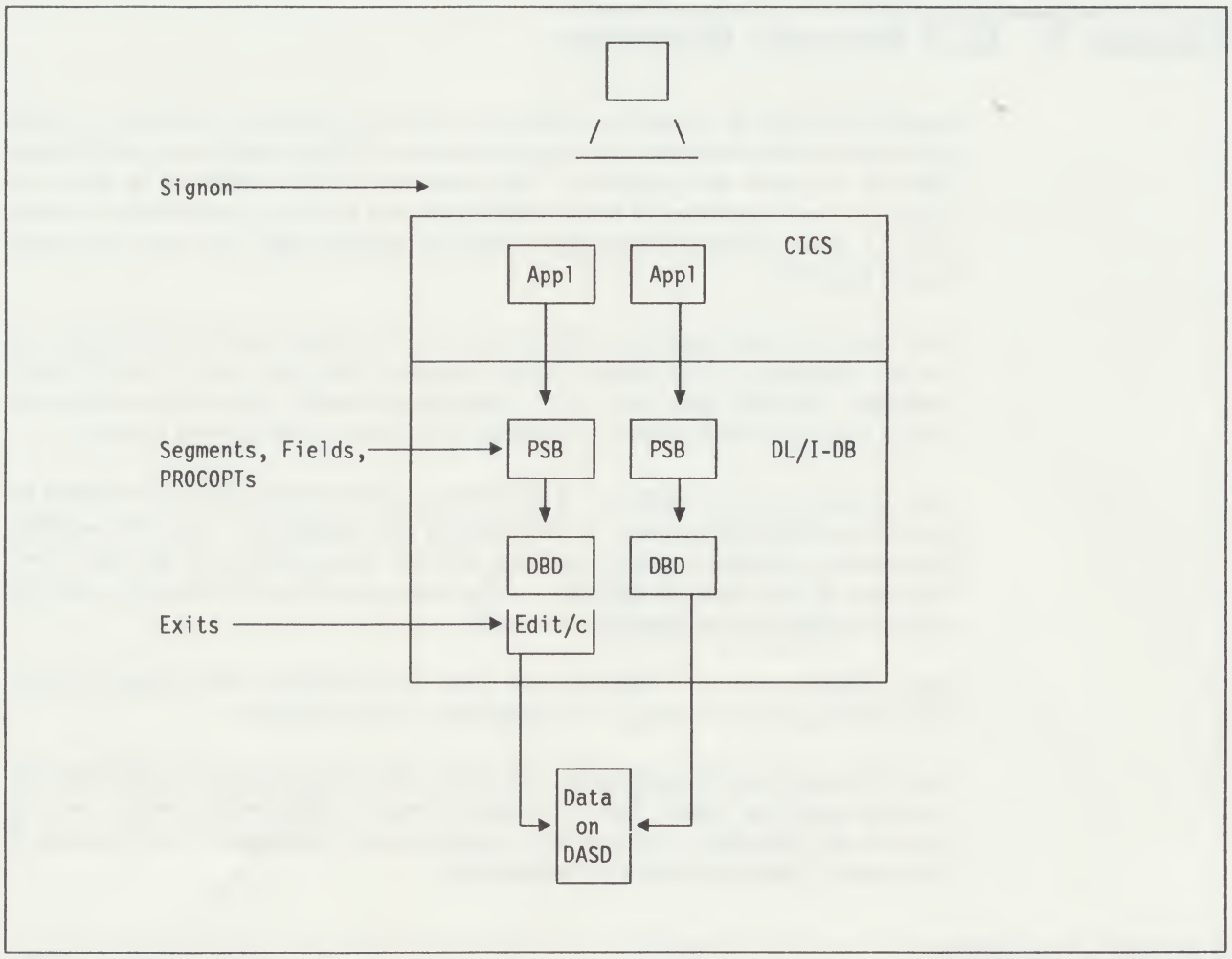


Figure 12. Outline of DL/I security

9.2 SQL/DS Security

SQL/DS security differs from DL/I security in that SQL/DS is always a separate subsystem. This means that all users, including batch, interactive, and utilities, must pass through SQL/DS security checking.

Security may also be applied to SQL/DS data at different levels:

1. The ability of any user or subsystem to connect to a given SQL/DS subsystem can be controlled. For example, you could prevent a test CICS subsystem from accidentally connecting to a production SQL/DS subsystem, or a legitimate user of the SQL/DS query and decision support subsystem from connecting to the SQL/DS subsystem used for online transaction processing.
2. CICS can control which USERID is allowed to run a given transaction and program. This may be accomplished by the CICS internal security system. The transaction manager or batch job can also pass the identity of the user to SQL/DS if required.
3. An SQL/DS package is needed to execute any SQL statements against any SQL/DS data. RUN authority must be granted to use a given SQL/DS

package. RUN authority may be given to a specific USERID, a list of USERIDs, or to PUBLIC (anyone), Until that RUN authority is granted, no user can run the program which uses that package. The SQL/DS programs may utilize static or dynamic SQL, or both.

- **Static SQL** is used where the SQL is coded inside the program. For static SQL, authorization to perform a given SQL function (select, insert, update, delete) is checked when the program containing the SQL statements is preprocessed, and the package is created. RUN authority is all that is needed by the user of the application to run the static SQL within the package.
- **Dynamic SQL** is used where the SQL statements are not known when the program is created. Dynamic SQL is typically used for ad-hoc queries. Authorization to run the package is needed, but in addition authority to execute each SQL statement is checked dynamically at execution time.

4. An SQL/DS view can be defined over one or more base tables or other views.

   The view can provide protection at the row level, column level, or even at the column value level. Any table or column not specified in the view cannot be accessed using that view. A user or list of users can be granted access to the view with the SQL GRANT statement. Users can be granted use of the view without being permitted to use the base tables referenced by the view.

   The SQL/DS view is somewhat similar to a DL/I PCB. It serves as a funnel into SQL/DS tables much as a PCB does for DL/I. The view, however, is much more powerful than a PCB:

   - The view can join multiple tables; the PCB has no such capability.
   - The view is used by all users, interactive, batch and programs; the PCB is part of the PSB which is related to programs only. Thus the view can be used as a security mechanism for all users not just for programs.
   - The view can limit access based on column value; the PCB cannot.

5. SQL/DS field procedures can be used to encrypt very sensitive data before storage in SQL/DS tables.

Figure 13 on page 156 shows an outline of SQL/DS security.

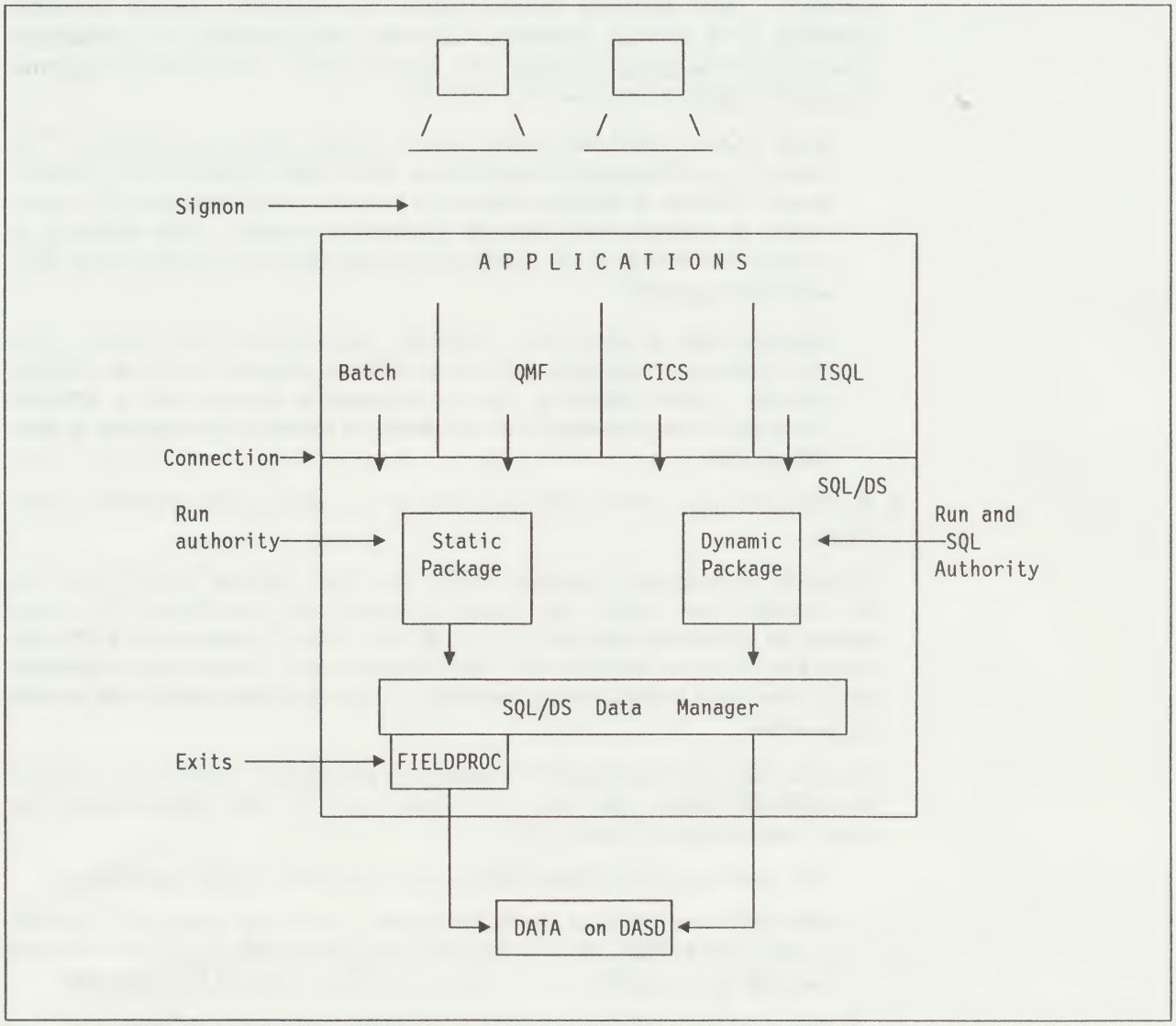


Figure 13. Outline of SQL/DS security

9.2.1 Other Aspects of SQL/DS Security

SQL/DS allows access from several different users and different types of users including ISQL. The ISQL interface provides an easy way to run SQL/DS, to inquire about SQL/DS status or various types of information contained in the SQL/DS catalog, as well as running interactive SQL queries. This interface is very useful for the maintenance staff as well as interactive ad-hoc users. As with any dynamically executed SQL statement, SQL/DS checks whether the user has the needed privileges. If a specific user has not been granted authority to access a given table or view, SQL/DS will refuse access to that table or view.

To aid in managing security SQL/DS has three type of authorities:

- **CONNECT** authority allows a user to connect to SQL/DS. A user must have connect authority in order to perform any database functions.
- **RESOURCE** authority is used to ensure that only properly authorized users can use space (resources) in the database.

- **DBA** authority is the highest level of authority in SQL/DS. If granted DBA authority, a user automatically receives **RESOURCE** and **CONNECT** authority. In general, a user with DBA authority is not affected by the SQL/DS authorization mechanisms. Such a user can perform all operations on all tables. In addition, users with DBA authority can perform certain functions that other users cannot perform. It is dangerous to give an untrained user DBA authority.

These authorities are very important. However, because of the power available to any user with this authority, the use of these capabilities must be carefully controlled.

It is this aspect of SQL/DS security which needs to be considered most carefully when migrating from DL/I to SQL/DS. Most of the system functions in SQL/DS can be performed interactively with ISQL. There is no equivalent access to DL/I data from an online terminal. Thought should be given to who will have access to maintenance or administrative USERIDs and under what circumstances.

9.3 Migrating Security

SQL/DS allows tighter control of security than is possible with DL/I. However, as with the other aspects of migration, minimum change is recommended to ease migration to SQL/DS. The security can be enhanced at a later time if required.

One sequence which could be followed is:

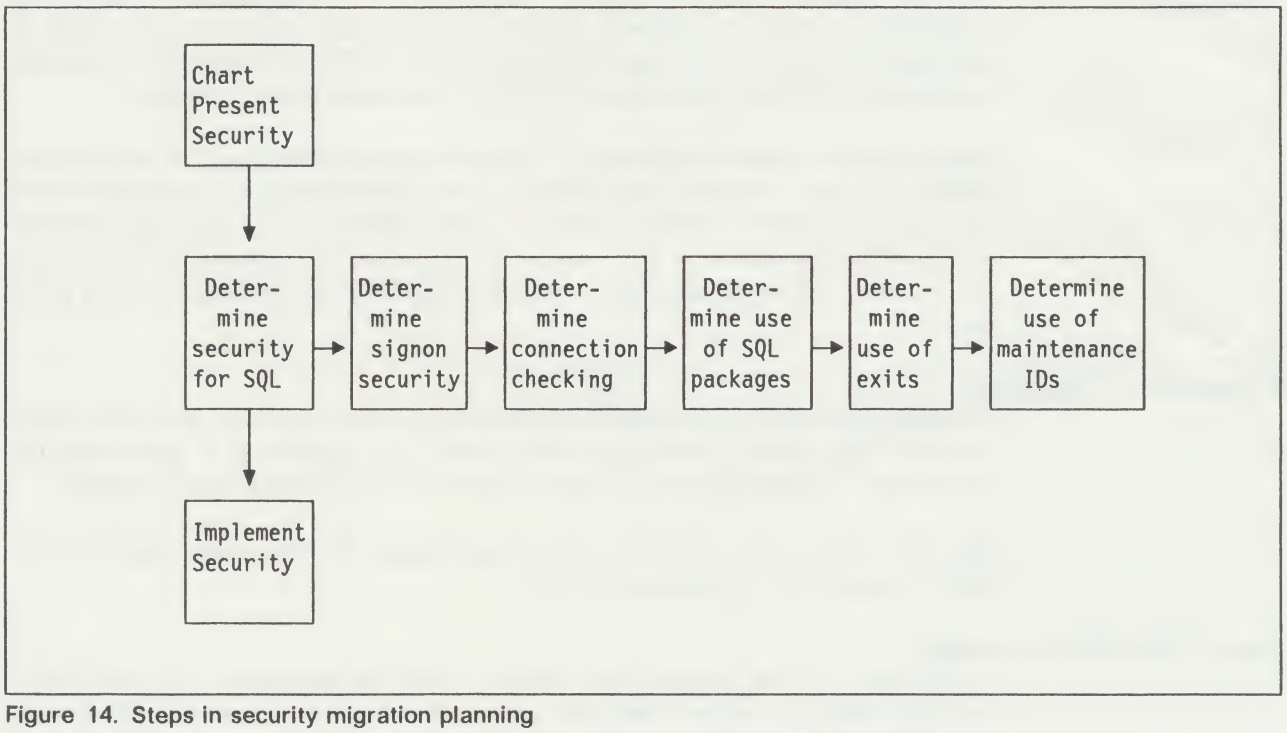


Figure 14. Steps in security migration planning

We will consider each aspect of migration in turn.

9.3.1 Chart Present Security

Unless present security is minimal, it will be beneficial to spend a little time documenting the present security so that it can migrated to SQL/DS.

It is suggested that a table with the following columns will be the most useful way to understand what is needed.

User IDs	Applica-tion	PSB	PCBs	PRO-COPTs	DBD exits	SQL/DS package	Run granted to	SQL/DS exits

This table can be used to relate the old to the new, for example, which new SQL/DS packages will replace which PCBs, and whether one or more is needed. If the User IDs are to be changed, then other columns may be needed.

9.3.2 Determine Security for SQL/DS

Signon Security

In the case of CICS, signon security is very likely to be unchanged and will work well with SQL/DS. In most cases, SQL/DS access to programs and data may be granted to PUBLIC and user access to transaction programs (and thus to data) can be controlled through the connection to CICS. Granting RUN authority to either the CICSID or to a list of USERIDs is normally recommended.

What should be added at this point is grouping for maintenance IDs and access control. Clearly, access to SQL/DS should not be available to just anyone who happens to invoke the necessary ISQL panels. Access to the packages that are used for the maintenance IDs should be restricted to those who really need this access. To avoid accidents, separate USERIDs for production are a good idea.

Connection Checking

Although this may be an extension of present security, it is well worth the effort involved. This means providing a list of who can connect to a given SQL/DS subsystem. In the case of CICS only the ID of that CICS subsystem is added.

This will ensure, for example, that programmers in test mode cannot accidentally connect to production SQL/DS.

Use of SQL/DS Packages

In this case we are mapping DL/I PSBs to SQL/DS packages. As one PSB is used for one transaction, there will generally be a one to one alignment but there may not be an exact match.

It may be that during migration, some function was moved to another program, or a portion of the transactions have been restructured resulting in more or

fewer packages. This needs to be considered only if CICS application packages are not granted to PUBLIC, as the question then arises as to which userids should obtain RUN authority.

Sometimes the same DL/I applications can be called via more than one transaction with its associated PSB. This may be because the users of the different transactions have different authorities. For example, the supervisor may have update authority but the ordinary users can only look at the data. One way to control this in DL/I is to have different PCBs with different PROCOPTs. The supervisor's processing options allow update, but if another user attempted to change something, DL/I would check the processing option, see that change was not allowed on the user's PCB, reject the call and issue a return code.

With SQL/DS, there is no direct correlation of this function as authority to execute an SQL/DS package includes the ability to run any static SQL within the package. There are three ways in which a migration of this function may be tackled.

1. Use dynamic SQL calls for the updates.

   This will make the conversion fairly straightforward as it allows each call to be changed one for one. However, dynamic SQL consumes more CPU resource than static SQL and it may be difficult to get the transaction to perform well.

   If this is done, authority to execute any dynamic update call will be checked at run time. This means that the USERID must be passed to SQL/DS from CICS.

2. Move all update calls into a separate module

   Normally, DL/I calls and SQL statements are written in line with the code but applications can use a 'call handler' routine to interface with the DBMS. If the update calls are all moved to a separate module, two copies of the application and two packages would be created. In the copy of the application written for the non-updating users, a dummy module would be added which simply issued a return code and/or message.

3. Rethink and rewrite the application.

   Depending on the need for this type of application, and the number of applications affected, this may or may not be the best solution.

Exits

The table drawn up will identify any exits currently in use. These should now be reconsidered as to their usefulness. There are a number of options:

1. Decide there is no longer any need for the exit, and scrap it or put the function in the application. In some cases the function of the exit may be accomplished without user coding. For example, if the exit is performing range checking on a specific field, or verifying that the field's data is in a list of permissible values, that may be accomplished through a view with check option.

2. Convert the exit to a FIELDPROC. These exits perform a similar function to DL/I DBD exits except that the FIELDPROCs operate on a column value.

3. Scrap the user-maintained exit and acquire an external tool. This particularly applies to compression routines.

Maintenance USERIDs

A maintenance USERID is needed to perform certain functions necessary to the care and feeding of the SQL/DS database system. The maintenance USERID is usually given **DBA** authority because one who holds this authority may perform operations that are otherwise unauthorized, for example issuing SQL Data Manipulation Language statements against an inactive table.

A user with **DBA** authority should be very cautious in the use of these special powers. The set of users who have this authority should be carefully controlled.

9.3.3 Implement Security

Security may be implemented with these steps:

1. Plan requirements early

   As security considerations may possibly affect the design of the migrated system, it is essential that security implications are considered and taken into account early in the migration process.

2. Work out lists of authorities and test plans.

   The exact grants should be listed[14] as, if no errors occur, they may be used to determine authorities on the production system.

3. Issue the minimum authorities needed at system test time.

   The time to implement and test the chosen security is during system test. All GRANTs should be issued on an SQL/DS test system. Anything missing or wrong could be identified by the test program.

4. Issue the same grants in batch on the production SQL/DS subsystem.

---

[14] You may wish to consider maintaining this list of GRANTS in an ICCF dataset or a CMS file which can be used by the Database Services Utility (DBSU) to assist in promoting an application from a test SQL/DS system to a production system.

Chapter 10. Coexistence Strategies

During the migration process, there will be data and programs that need to span both DL/1 and SQL/DS databases. In some instances, coexistence will be an interim step to full migration. In other cases, coexistence may be a way of life for a longer period of time. Therefore, a plan for dealing with coexistence issues is described in this chapter.

There are a number of different approaches or strategies for coexistence. The level of complexity and applicability of these strategies varies depending on the application requirements.

This section covers various types of coexistence, beginning with the simplest and proceeding to more complex. Most installations will find that as they proceed with the migration process, they will be implementing several variations of coexistence, depending upon business situations, application types and use of the data.

10.1 Extract Data for Query, Decision Support Systems

Extraction for query and decision support need not involve migration per se of applications or data. If the initial benefit of implementing SQL/DS is to obtain the capability of using the relational DBMS in support of end-user capabilities such as query and decision support, extract is an appropriate approach. In this case, it will still be necessary to go through a database design effort.

Extract implies a periodic nature. This means that the extract tool must have adequate performance characteristics. See Chapter 6, "Data Migration" on page 77 for discussions of issues, tools and approaches in extracting data. Note also, however, that once initial extraction has been done, an incremental update technique may be an alternative to periodic extraction; this is discussed below in 10.3, "Synchronization of Data Between Systems."

If the extracted data are to be processed in a read-only manner, there are no further issues. If the extracted data are to be updated independently of the production data, there is the question of synchronization, which is covered later under 10.3, "Synchronization of Data Between Systems" on page 164. If the data are not to be synchronized, but rather the updates to the extracted (relational) data are changes appropriate only to the end user (such as 'what if?' cases), there may be a concern that these updates will be lost due to a subsequent extract, if that extract replaces or refreshes all of the relational data. In such cases, an appropriate approach would be to prohibit changes to the directly extracted data and require table extensions, or new tables, for relational data that are derived and manipulated. The users would then be responsible for procedures reflecting the newly extracted data in these additional tables. Note that a similar approach can be used to support individual users' variations. While this provides a more complex environment, it is one which the relational model readily permits due to its extensibility.

10.2 Programs Requiring Data Stored on Both Systems

In an integrated database environment, it is not possible to say that each data element is used in only one application. Consequently, as applications are migrated, cases will arise where a program requires data from both database systems. There is no restriction on accessing data in both SQL/DS and DL/1 from a single program, but there are some considerations.

Programs scheduled for migration that will continue to require access to DL/1 data will have to be given special consideration. Those data requests that will continue to access or modify DL/1 data should perhaps be prepared for future migration. If the DL/1 data used by the program is not to be migrated immediately, but *is* planned to be migrated within the next few years, it could be desirable to isolate the application logic that may be migrated in the future into subroutines, to permit change at a later time with minimal program redesign. This may increase the current migration effort by some small amount, but reduce future migration resource requirements.

On the other side of the issue is the program that is part of an application system not being migrated, but now must access some SQL/DS data. This would require modification of that program to replace some DL/1 logic with SQL/DS logic. This is often not perceived as part of the migration, but must be included in the cost and effort. Consideration should be given to alternatives: changing the program by making a minimal change of data access logic to support only the data that was migrated; preparing for future migration as described above; or duplicating (replicating) the data (storing that particular record or row in both DL/1 and SQL/DS) and avoiding for now any change to the application. Individual circumstances such as the amount of data logic against each database, the timing of migration of the application, program complexity, etc., need to be considered.

10.2.1 Programs Updating a Single DBMS

The most desirable condition of access to data in both DBMSs is where data in at most one DBMS (DL/1 *or* SQL/DS) is updated, but not in both DBMSs within the same unit of work or transaction. In this case, there are no issues beyond the code migration described above. Whichever database has been updated will be responsible for its own integrity.

10.2.2 Programs Updating Both DBMSs

If a program has a requirement to update data in each of the DBMSs, there are potential integrity considerations. In order to maintain consistency between the two databases, a commit coordinator is required for any program making changes to the databases. Programs should be reviewed to ensure that the synchronization points recognized by the coordinator are consistent with application requirements.

For online transactions under CICS, coordinated recovery for DL/I and SQL/DS is provided by the transaction manager. The CICS Two-Phase Syncpoint (TPSP) protocol provides for consistency among multiple resources. The process ensures that all CICS resource adaptors are consistent in their commit work or rollback activity. For example, a transaction making changes to DL/I resources, SQL/DS resources, and CICS resources would either commit the changes for all the resources or would be requested to rollback all the changes. Thus, related

data, across multiple SQL/DS and DL/I databases can be kept in a consistent state.

Programs should be reviewed to ensure that the synchronization points are consistent with the application requirements. The TPSP protocol is driven at points which end a CICS logical unit of work and an SQL/DS logical unit of work. At these syncpoints, the CICS syncpoint manager requests all resource adapters accessed by the application to either commit or rollback the updates (if any) to their respective databases.

Syncpoints occur during the execution of an application because of the following:

- An application issues an EXEC CICS SYNCPOINT (COMMIT or ROLLBACK)
- A CICS transaction terminates normally (resulting in COMMIT) or abnormally (resulting in ROLLBACK)
- The SQL COMMIT WORK or ROLLBACK WORK statement is issued, causing a CICS SYNCPOINT (COMMIT or ROLLBACK) to be issued The SQL ROLLBACK could be issued internally, as a result of the SQL/DS operator command FORCE.

To take advantage of the coordinated recovery between SQL/DS, CICS, and DL/I, the CICS installation must include the following:

1. DBP=YES or DBP=xx in the CICS System Initialization Table (DFHSIT)
2. DTB=Yes for each online application program that accesses SQL/DS. This is specified in the assembly of the CICS Program Control Table (DFHPCT).

For batch programs, including those executing under MPS, there is no commit coordinator between the two DBMSs; consequently, there could be a failure following the commit of data to one database but before the commit to the second. The result would be data committed in one database and backed out in the other. The degree of business risk must be evaluated.

Assuming that the second commit statement immediately follows the first, there should be no potential for application failure between the statements once the program goes to production status. The potentials for failure, therefore, are in one of the two DBMSs, in the operating system, or in the hardware. As all of these are generally stable environments, the probability of a failure occurring between the execution of two commands is very small, but still possible. Discovery of the condition can be made by interrogating the logs for the two systems when it is known that such a transaction was in progress at the time of the failure. This is not a simple task, and detailed procedures would need to be developed in advance.

10.2.3 Parent-Child Relationships Split Between DL/1 and SQL/DS

Again due to the degree of integration of DL/1 databases, it can be anticipated that at some point, the condition will arise that as data are migrated for an application, data that are hierarchically related to the migrated data will remain in DL/1. Database design considerations for this are included in 5.5, "Designing for Data Related Between DL/I and SQL/DS" on page 71.

Programming considerations fall into several categories. The first is the case where the program updates both segment types in the relationship: parent and

child. In this case the problem is essentially the same as described in 10.2.2, "Programs Updating Both DBMSs" on page 162 with the addition of the requirement for programmed integrity control. If the child is the part of the relationship that is now in SQL/DS, the program which inserts the child will now be responsible for verifying that the parent(s) exist in the DL/1 database and for building the equivalent of a foreign key which will permit identification of the parent. Programs that delete the parent must enforce the equivalent of the delete rules and either delete the child row in SQL/DS or set its 'foreign key' to null.

If the part of the relationship that has moved to SQL/DS is the parent, and the implementation in the database is to replace the DL/1 parent segment with a dummy, insert logic will have to be extended to support adding the DL/1 dummy parent segment occurrence at the same time the SQL/DS row is inserted. In the case of deletions, the application must delete both the SQL/DS data and the dummy segment occurrence. Programs that access the child in DL/1 may have to be modified to accept the dummy parent segment occurrence in place of the full segment.

The second condition that must be addressed is where the program is not aware of the data on one of the systems, but performs updates which affect it. This will primarily be the case where the program deletes data which was a parent of one or more children, but did not have access to the child segment(s). If that parent is moved to SQL/DS, it is now necessary for the application to access segments (the children) in DL/1 to cascade the delete. Every program which has delete capability against a parent, and which will be migrated, should be analyzed to determine the degree of impact. The least obvious case will be where the parent has not been replaced with a dummy in DL/1, but with "foreign keys" in the DL/1 child segments.

10.3 Synchronization of Data Between Systems

All of the above discussion of coexistence assumes that the same application program operates against both databases and that there is a single copy of the data. In some cases it will be expedient to have the data replicated under both DBMSs, or it might not be possible for a single program to update both DBMSs. These conditions require concern for synchronization of the data under two DBMSs. This condition is not unique to a database environment. For example, tape systems have had the same problems for many years, and a number of the alternatives are the same.

If the data has been replicated, the reasons for the replication probably negate updating both copies with the same program. If, however, it would be appropriate to update both copies, the considerations under 10.2.2, "Programs Updating Both DBMSs" on page 162 apply.

The other important condition is where it may not be appropriate to have a single program update both systems. In this case it is necessary to *propagate* the update in some way to the second system. If the data are replicated, several options exist to update the replica. They vary from "trickle update" to mass refresh. Some are asynchronous forms of synchronization, as contrasted with single program update.

Other user-implemented techniques can be considered. For example, a user-generated change file can be generated for mass refresh of the replica. This is easily done with a programmed journal of changes in the case of updates generated in the new SQL/DS program. In the case of the existing DL/1 program it will be necessary to change the code to provide a subroutine to generate the journal and subsequent mappings to SQL/DS. The standard DL/1 log cannot be used for this purpose, since the log data does not include the fully concatenated key necessary to map a segment occurrence to its equivalent row in SQL/DS. The use of a programmed journal of changes will again mean that a program not directly affected by the application being migrated will have to be modified. The degree of impact will depend upon the number and variety of updates in the program.

With the mass refresh approaches described above, updates are delayed by a period of time until they are applied to the receiving system (often by a batch job). A 'trickle update' approach may provide more timely synchronization. This technique is amenable to both replicated and related data. The application program that updates the data must initiate a transaction which will cause the corresponding update on the other system. If a transaction program already exists in the current system to perform such an update, it may be usable as a basis or starting point for the 'trickle update' program. Normally, however, there will need to be a special transaction processing program to isolate the updates. The transaction can be initiated via communications link and processed as resources become available. The transaction could also be a batch file or job that will be processed as batch programs are scheduled. If the updating program is on the SQL/DS side, it is fairly easy to add a subroutine to do this, the usage of which may be discontinued when the DL/1 data are migrated. If the program is on the DL/1 side, it must be changed and logic added.

Replicated data are most easily managed when updates are driven from one DBMS or the other. Where common data are being updated on both systems, there is always the possibility that updates might be lost. For example, a program on one side might delete the data being updated by the other side. Such conditions should be *avoided* if possible in the database and application designs, as well as in the overall system design and planning for a database coexistence environment.

Chapter 11. Operations

11.1 Operational Considerations

Although SQL/DS appears different operationally, migration of jobs and utilities to run with SQL/DS instead of DL/I is quite straightforward.

Operations will have to learn to manage SQL/DS, since it is started and stopped independently of CICS. In fact, CICS can fail and SQL/DS may continue to process requests from a batch job or CMS user.

This means that operations will have some new procedures to write to deal with SQL/DS as a database management system, and some utility and batch jobs to review and change, but essentially, migration to SQL/DS should be fairly straightforward as far as operations are concerned.

11.2 New Operational Items

There will be new activities required for SQL/DS transactions and jobs to function normally.

1. Train operators about SQL/DS. The messages issued by SQL/DS are, of course, different and the procedures to be followed will also be different.
2. Write new operator procedures. A sample list is:
   a. How to start SQL/DS normally.
   b. How to stop SQL/DS normally.
   c. How to respond to selected SQL/DS messages.
   d. What to do if SQL/DS will not start.
   e. What to do if SQL/DS will not stop normally.
   f. What to do if an SQL/DS batch job fails.
   g. What to do if an SQL/DS utility job fails.
   h. What to do if SQL/DS issues I/O error messages.
   i. What to do if an archive job fails.
   j. What to do if SQL/DS fails.

   See *SQL/DS System Administration* (VM: GH09-8084, VSE: GH09-8096) for more information on procedures and commands pertaining to the above operations.

11.3 System Backup and Recovery

Both DL/I and SQL/DS have utilities and recommended procedures which are used for saving and recovering the database information. Each installation may have site specific procedures in place. Since backup and recovery procedures are closely coupled with the specific DBMS, there is not much information in this area which is specific to the conversion process. However, it is necessary to develop and test a backup and recovery plan.

Backup and recovery is SQL/DS is discussed in detail in the *SQL/DS System Administration* (VM: GH09-8084, VSE: GH09-8096)

ARCHIVE is the SQL/DS tool provided to protect your entire database, the directory, as well as the data extents. There are three types of archives:

- A database archive includes both the database directory and extents.
- A log archive backs up only the log.
- A user archive is an archive of the directory and extents created by a non-SQL utility (e.g. VMBACKUP or DDR).

The LOGMODE parameter, specified at SQL/DS startup governs the archiving and level of recovery capability supported. It specifies whether or not an archive file is to be kept for the database, or the logs.

LOGMODE=A Database archive, full recovery possible

LOGMODE=N No recovery, for single user mode only

LOGMODE=L Log archive, database archive, full recovery possible

LOGMODE=Y Database archive, no forward recovery

The output from the database archive and the log archives will be the input to SQL/DS to start recovery. Recovery, like backup, is controlled by an SQL/DS startup parameter.

STARTUP=R Uses backup data from SQL/DS ARCHIVE

STARTUP=U Uses backup from non-SQL/DS user sources.

LOGMODE=L Uses current log, plus database and log archives

LOGMODE=A Uses current log plus last database archive.

Careful consideration must be given to the types of archives and the methodology for recovery. These will vary greatly depending on database size, volatility, and time available to backup and restore.

See the "Recovering from Failures" section in the *SQL/DS Database Administration* (VM: GH09-8083, VSE: GH09-8095) for a comprehensive discussion of backup and recovery for SQL/DS.

An IBM publication, *An Operational Archive System for SQL/Data System (Version 1) Release 3.5 under VM/SP* (GG24-3177) describes a prototype archive system and includes sample programs to show how to automate backup and recovery of SQL/DS using tapes. This manual refers to an unsupported level of SQL/DS, but the information is still appropriate.

The IBM program offering 'SQL MASTER/VM' automates many operational tasks, including archiving. It supports all logmodes, and greatly facilitates recovery.

In addition, SQL/DS offers a series of functions under the DBSU utility that may be used to backup or restore specific tables. These will be discussed in a following section.

11.4 Job Control and Processing

DL/I job types and specific job control language (JCL) in jobstreams require analysis to be able to derive the equivalent SQL/DS jobstreams, but the migration is straightforward for the majority of the JCL statements.

System level jobs are mentioned here but not addressed in any detail. For more information, refer to the *SQL/DS Database Administration* (VM: GH09-8083, VSE: GH09-8095) and the *SQL/DS System Administration* (VM: GH09-8084, VSE: GH09-8096).

11.4.1 CICS Start Up

The DL/I and SQL/DS versions are essentially the same, assuming that the appropriate installation instructions regarding the CICS table entries are followed. The SQL/DS installation manual supplies the file statements that must be added to the CICS startup job stream.

11.4.2 DL/I Batch

A DL/I batch job that executes in its own partition under the control of DL/I could be replaced by an SQL/DS job in "single user mode". In this mode, only a single application or utility can perform against the database.

11.4.3 DL/I "MPS" Batch

This batch environment is functionally similar to the "multiple user mode" in SQL/DS. Multiple user mode is the standard way of operating SQL/DS. SQLINIT must be executed by each user, with SQL/DS residing in its own partition or machine. Unlike DL/I, however, SQL/DS in "multiple user mode" does not require CICS.

DL/I Utilities

The DL/I utility programs and their SQL/DS equivalents are discussed later in this chapter.

11.4.4 Application Job Control Language (JCL)

Invoking applications under SQL/DS will vary according to whether single user mode or multiple user mode is selected.

The job control for single user mode executes the SQL/DS program and passes it the name of the application, which SQL/DS will then invoke.

For multiple user mode, SQL/DS must already be initialized in its own partition. The pre-processed application program is executed in a separate partition using standard job control.

Error Handling: Jobstreams which use the COND parameter of the EXEC statement may be affected by the conversion, as DL/I automatically sets the system return code to 8 when a program terminates abnormally. SQL/DS does not do this automatically; it is the application programmer's responsibility to set this code.

DLBL or EXTENT or FILEDEF

As with any jobstream, these statements will be specific to the program needs. There are several general changes that will apply, however.

- The LIBDEF statement will need to be changed to specify which of the of the libraries to search, which to use for cataloging, etc. VM jobstreams will need to use the FILEDEF statement.
- A FILEDEF or DLBL statements must be supplied for SQL/DS batch jobs. This statement permits the inclusion of commands and subcommands. For more information on these commands, see the *SQL/DS Database Services Utility* (VM: SH09-8088, VSE: SH09-8100).

Depending upon your environment, the user is referred to the VM or VSE version of the SQL/DS operations guide for detailed information about specific parameters.

11.4.5 SQL/DS Utility Job Control Language (JCL)

DL/I utility jobs execute the utility program directly, supplying the necessary file definitions and input parameters particular to the utility. The SQL/DS Database Services Utility (DBSU) is invoked using DLBLs or FILEDEFs to supply the necessary file definitions with an input stream containing the commands for the utilities. See the IBM *SQL/DS Database Services Utility* (VM: SH09-8088, VSE: SH09-8100) for the required syntax and general considerations.

SQL/DS utilities run as a standard VSE or CMS application program. Users of VM may create standard EXECs to invoke the utilities.

Often there are many sets of JCL or EXECs maintained for an DL/I utility to allow for each variation either in parameters or applications. Each can be converted to an equivalent set of JCL or EXECs for SQL/DS. An alternative is to maintain SQL/DS utility commands separately from the execution JCL or EXECs.

11.5 Utilities

The major DL/I utilities are listed below with their applicable SQL/DS counterparts.

11.5.1 Database Utilities

Due to the differences between the structures and, therefore, the maintenance of SQL/DS and DL/I, it is impossible to provide a one-to-one relationship between the respective utilities.

The following table is a functional comparison showing the DL/I utility programs and their SQL/DS equivalents, if any:

NAME	FUNCTION	SQL/DS EQUIVALENT
DLZURGU0	HD reorg unload	DBSU unload, dataunload
DLZURGL0	HD reorg reload	DBSU reload dataload
DLZURUL0	HISAM reorg unload	DBSU unload, dataunload
DLZURRL0	HISAM reorg reload	DBSU reload, dataload
DLZPRCTn	Partial DB reorg	DBSU DBSPACE/table unload, reload
DLZUDMP0	Data set image copy	Database archive
DLZUCUM0	Database change accumulation	Log archive
DLZURDB0	Data set recovery	STARTUP parameters
DLZBACK0	Database backout	STARTUP parameters

Table 18. DL/I Utilities. DL/I utilities and equivalent SQL/DS functions.

NAME	FUNCTION
DLZURPR0	Pre-reorg
DLZURGS0	Scan
DLZURG10	Prefix resolution
DLZURGP0	Prefix update
DLZLOGP0	Log print utility

Table 19. DL/I Utilities. DL/I utilities that do not have an SQL/DS equivalent and are not required for SQL/DS.

11.5.2 Data Handling Utilities

The data handling utilities in DL/I, specifically DLZURGU0, DLZURGL0, DLZURUL0, and DLZURRL0 are replaced by a single utility in SQL/DS called the Database Services Utility, or DBSU. DBSU is an SQL/DS application program that allows the user to specify parameters that perform the following:

NAME	FUNCTION
DATALOAD	Load tables from sequential file
DATAUNLOAD	Unload tables to sequential file
UNLOAD	Unload tables to SQL/DS format
RELOAD	Reload tables from Unload format

Table 20. DBSU facilities. Commands available to load and unload data to/from SQL/DS.

In addition, DBSU can also be used to process SQL commands such as CREATE table or index, as well as to unload/reload access modules (packages) to/from the database via the UNLOAD/RELOAD PROGRAM commands.

11.5.3 Database Recovery Utilities

The DL/I database recovery utilities, DLZUDMP0, DLZURDB0, DLZBACK0 or DLZUCUM0 are replaced by the previously discussed ARCHIVE function. Note that this function is determined by the SQL/DS startup parameters.

11.5.4 Operator Commands

SQL/DS provides several operator commands that display valuable information about the status of the database.

- COUNTER * describes the activity of the database since the last RESET * command. It is used for monitoring buffers, locks and I/O activity, as well as providing other useful information.
- SHOW commands (ACTIVE, DBCONFIG, DBEXTENT, DBSPACE, LOCK, LOG, SYSTEM) are used to monitor specific areas of the database.
- TRACE commands (TRACE ON, TRACE OFF) provide a method of tracing SQL/DS. Programs are provided by SQL/DS to print the trace results.

11.5.5 Data Definition and Access

This section discusses DBDs, PSBs, ACBs and the SQL/DS equivalents, DDL and application packages

DL/I uses a number of definition utilities:

DBDGEN This utility creates a Database Descriptor (DBD) which defines the layout of the segments and fields.

PSBGEN This utility creates a Program Specification Block (PSB) which defines which segments of a database (as defined by a DBD) a program may access and what actions it may take, i.e. retrieve (get), update, delete, replace, load.

ACBGEN This utility creates an Access Control Block (ACB) which is a combination of a given PSB and DBD. Creation of this helps on-line performance as it saves building it each time a transaction runs.

SQL/DS does not use control blocks in the same way. The nearest equivalent to a DBD is the SQL/DS SQL data definition language (DDL). SQL/DS DBSPACEs, tables and columns may be defined using the ISQL interface.

The nearest equivalent of the PSB and ACB is the SQL/DS application package or access module. To access any data, a program uses SQL data manipulation language (DML). Programs using static SQL (those where SQL can be predefined) are prepared through a precomplier to produce a an access module or package. The package effectively contains SQL and paths to the data.

The precompiler is executed as part of the program preparation process using the DBSU. For more information, reference *DB2 Application Programming Guide*, (SC26-4377).

At run time, the application is associated with a particular plan and permission to execute that plan is granted to a USERID or to PUBLIC (anyone).

11.5.6 Housekeeping

This section describes those utilities which must be run from time to time, but which are not essential at any one time. These utilities must be run to keep the SQL/DS catalog and data consistent and in good shape.

UPDATE STATISTICS This utility updates the SQL/DS catalog with DBSPACE and index space statistics such as cardinality of data (row and index column occurrences of values) and clustering, to provide information to aid path selection during the PREPARE process.

SQLCIREO This utility reorganizes the index on catalog tables.

11.5.7 Report Utilities

ISQL, an interactive EXEC under VM or a CICS transaction under VSE, provides formatting both for display and printing. This EXEC or transaction is part of the SQL/DS product.

Query Management Facility (QMF) is one of many available programs that offer a more sophisticated and user-friendly report generator, as an alternative to ISQL or a user-written application program. Reference *SQL/DS Applications, Tools, and Services* (GX09-1218) for a more complete list of application programs.

Appendix A. Hierarchical Database Architecture

This section provides a brief high level overview of the hierarchic architecture and DL/I data structure. It is intended for the person with little or no DL/I background who may be involved in migration or coexistence with DL/I and SQL/DS. Italicized words in the text are those which are specific to a hierarchy or DL/I and are used elsewhere in this Guide. After completing this section, the reader might also read the glossary for additional concepts.

In a hierarchic database, data is represented as a structure, where certain information is related to other subordinate types of information in a hierarchical manner. This hierarchy is called a *logical data structure.*

DL/I application programs deal with logical data structures. *Logical* refers to the manner in which the application program sees the data. A logical data structure is always a hierarchy of data record types called *segments.* An application program in DL/I never deals directly with a physical data structure. Figure 15 on page 176 shows an employee database as it could be described in DL/I.

In the example, the structure depicts five different types of segments, each consisting of one or more data fields. A segment at the top of the hierarchy, that is, one not dependent on another segment, is called a *root segment.* A *database record*, by definition, consists of one root segment occurrence and all of its dependent segments.

In the example shown, the Employee segment would contain the identifying information for the data structure (e.g. name and number) and is the root segment.

There can also be many levels of dependency. For example, the Taxes and Deductions segments are subordinate to, and depend on, the Salary segment. The Salary segment, in turn, depends on Employee, the root segment.

The basic unit of data manipulation is the segment. The structure of the hierarchical database is defined via a *parent/child relationships* between segments of data. In our example, the root segment (Employee) is the parent of the segments (that is, Salary and Address) that depend on it. The dependent segments, Salary and Address, are called *children* of the parent segment (Employee). The *child* segment (Salary) can, at the same time be a *parent* segment; that is, have children itself (Taxes and Deductions).

There may be one or more segments defined as child segments for any parent segment. Also, for any occurrence of a parent segment, there can be zero, one, or many occurrences of a segment which is a child of that parent; notice that in the example there are three occurrences of the Deduction segment for the Salary segment shown.

Because each dependent in the hierarchy has only one parent (unless logical relationships are used), the hierarchical data structure is sometimes called a tree structure. Each branch of the tree is called a *hierarchical path.*

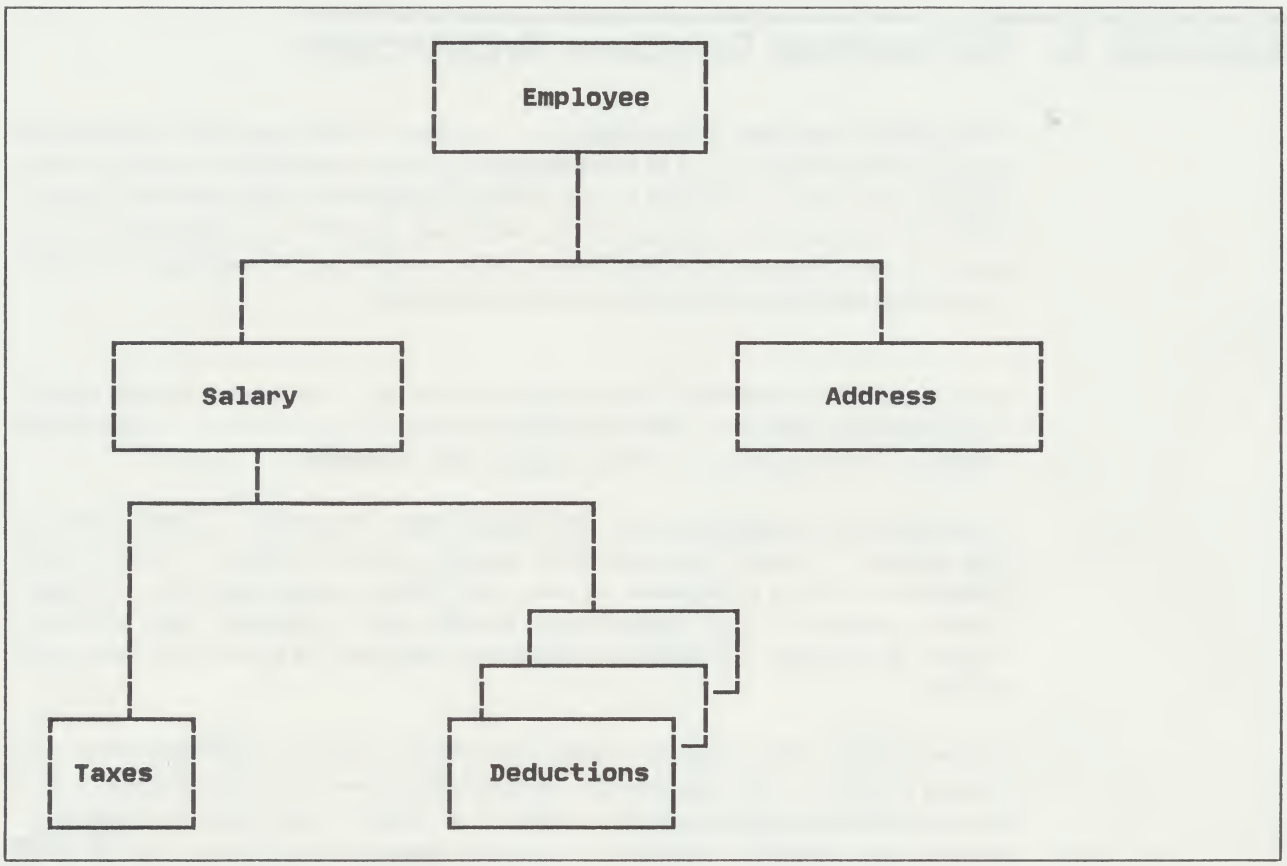


Figure 15. Logical database structure (hierarchy)

DL/I supports two physical storage organizations - *hierarchical direct* and *hierarchical sequential*.

In the hierarchical direct organizations, segments are stored in blocks, and are related by direct addresses (pointers).

Segments in the hierarchical sequential organization that represent one database record are related by being physically adjacent. This requires that segments representing one database record be contained in a variable number of storage blocks unique to that database record.

The logical and physical definitions of the data base are kept in libraries external to the programs in the DL/I system itself. These libraries include data base descriptions (DBDs) and program specification blocks (PSBs). A DBD describes a data base: the names of the segments, their hierarchical relationship, and the physical organization and characteristics of the database. The DBD provides DL/I with the mapping from the data structure of the data base as seen by applications to the physical organization of the data. The data structure can be re-mapped into a different physical organization without program modification. There are two types of DBDs:

The *physical DBD* provides the definition of a single hierarchical structure.
The *logical DBD* provides the definition of one or more related hierarchical structures into a new hierarchical structure.

The logical DBD relies on the logical relationships that were defined in the physical DBD(s). (Logical relationships may be used to provide an additional

access path to a segment, possibly from a segment in another physical DBD. Thus a limited form of support for a network data model is possible - a segment may have one physical parent and one *logical parent*.)

The PSB defines the data structure for each application program. More than one program may use the same PSB. Each PSB contains one or more PCBs (program communication blocks), one for each hierarchical (sub)structure the program sees of the data base. It specifies for each segment the kind of access allowed by the program (read only, update, insert, load, and delete).

Access to records stored directly may be by a randomizing routine (which provides a record address calculated from a defined key), by an index, or by a data base scan.

DL/I provides operators to retrieve segments (get), update existing segments (replace), add new segments (inserts) and delete existing segments.

The application programmer can request DL/I functions in one of two ways: through the High Level Programming Interface (HLPI) or through the lower-level program CALL interface.

In DL/I the program must navigate the data structure. For example, to retrieve the Deductions segment, the program must know that it is a dependent of the Salary segment. Because the application programs have to "understand" the data base structure, if we change the data base segment hierarchy, programs will have to be changed as well.

When a segment is deleted, all its dependents are deleted automatically. In the Figure 15 on page 176, if we were to delete the Employee segment, all the salary, taxes, deductions and address segments for that employee would be deleted as well by DL/I. The programmer does not issue delete commands for the dependent segment.

Appendix B. WITT

WITT (Workstation Interactive Test Tool) will identify inconsistencies which might be missed if users run transactions and manually look for failures. WITT may also be useful after batch execution, to interrogate representative samples of data to compare DL/I with SQL/DS.

WITT is an OS/2 Presentation Manager application that provides host and OS/2 Presentation Manager application regression test services. It saves recorded terminal sessions in programmable workstation disk files that can be sent to a host system for storage by means of a built-in file transfer function. Regardless of where stored, these recorded files (called test cases and screen image files) can be dynamically retrieved for session replay either singly or in combination.

Test case files contain keystrokes and/or mouse manipulation records and any logic functions the user may have injected during the record session. The screen image files contain copies of those application display screens pertinent to the test. WITT also has the ability to display a current screen different from its archive counterpart and highlight the area(s) that do not match (the two versions of the screen may be alternately displayed by pressing a function key).

The WITT user can display an archive screen and easily mark areas not critical for future comparisons. This means that a user when watching a run of test results can identify areas like time stamps which will always be different, so that only screen areas where application results really differ will be highlighted.

A WITT user testing host applications can create multiple variant test cases from one prototype test case by means of variable substitution. The host user can also create test cases with extended logical capabilities including decision making, and interval timing. All users can use the host or programmable workstation screen data capture.

A user can also set up a series of test case executions to be executed at a later time. This enables the user to more productively use machine resources, and maximize testing effectiveness.

WITT can generate both detailed descriptions of test case mismatches, as well as summary reports describing test case success and failure. It can also automatically generate IBM Bookmaster input files from saved screen images for print.

Appendix C. Sources of Assistance

This appendix lists several sources of assistance that may be useful in migration or coexistence with DL/I and SQL/DS. IBM, a number of IBM business partners and other third party vendors are identified. This list does not in any way constitute a complete or comprehensive list of vendors and/or tools that might provide assistance with migration and coexistence. Furthermore, these references are for information only and do not constitute a recommendation or endorsement by IBM of the third parties and their capabilities or products.

International Business Machines Corporation

150 Kettletown Road
P.O. Box 4001
Southbury, CT 06488-4001
(203) 262-2000

DBMS migrations are a management challenge, not a technical challenge.

The primary challenge faced by any organization considering a DBMS migration is identifying, obtaining and organizing a skilled project team. This team must include skills in: migration management, the source and target DBMSs, the application area, application testing, programming languages used and with any conversion tools utilized.

IBM is uniquely qualified to assist clients in DL/I-DOS/VS and SQL/DS migrations and coexistence. We have unsurpassed skills in both the source and target DBMSs, a time-proven migration methodology, industry-specific application skills and access to industry as well as internally developed computer-based tools. While the migration is in process, we can provide education as well as on-site training and consulting for your SQL/DS environment. Whatever is needed to assist in DL/I-DOS/VS and SQL/DS migration and coexistence, **IBM** can provide the best support in the industry.

IBM Migration Methodology

The **IBM** migration methodology is a process by which computer programs (and their data and documentation) are modified to be compatible with a new or significantly altered I/S environment. In this case, compatible means: **running in the new environment and yielding the same functional results.**

The migration methodology is divided into four phases:

- Analysis
- Pilot
- Conversion
- Cutover

Each phase has its own specific characteristics, objectives and deliverables. At the end of each phase, a plan for the next phase is prepared and the results are presented to client management for a "GO/NO-GO" decision. This process, developed and refined over a period of 15 years by **IBM** Project Mangers, has been critical to the successful completion of our client's migration projects.

IBM skills make the difference

Your goal is to have productive SQL/DS information systems working for your organization as soon as possible. One way to accomplish this goal is to have **IBM** skills as a key part of your team. Our Project Managers can take responsibility for the entire project including subcontractor performance or use our skills to supplement your project team - whatever best fits your needs. **IBM** skills are continually enhanced through one of the industry's most comprehensive training programs including advanced courses in the newest hardware/software systems, migration techniques, data base design and CASE-based application development. Our commitment to ongoing education and training enables us to deliver the best available skills to your organization. These skills can help assure you of a successful DL/I-DOS/VS and SQL/DS migration and/or coexistence project.

For more information on DL/I-DOS/VS and SQL/DS migration and coexistence services, contact your local **IBM Marketing Representative**.

ANDERSEN
CONSULTING
ARTHUR ANDERSEN & CO., S.C.

Andersen Consulting is a world-wide organization providing services in systems design and installation, systems integrations, system productivity consulting, information planning, strategic consulting, change management and facility/network management. In the fiscal year 1991, Andersen Consulting employed more than 21,000 consulting professionals world-wide, and forecasted revenues of $2.3 billion.

Andersen Consulting has the following qualifications relevant to DL/I and SQL/DS:

- Project management

  Method/1, Andersen Consulting's methodology and project management tool set, supports our world-wide organization in the management and control of critical systems development projects. Our Quality Assurance program helps assure the delivery of high quality systems on time and within budget.

- DL/I experience

  The Firm has extensive experience in designing and developing DL/I systems using a variety of development tools. Representative systems include system design/installation for a small equipment manufacturer, physical and logical database design of a DL/I-CICS customer information system for a major food chain, system review and performance tuning, (including analysis of the physical and logical database design, technical architecture, program design and DL/I utilization), for a manufacturing firm, etc. Andersen Consulting provides DL/I expertise to most industries, covering both, the native product and related development tools.

- SQL/DS experience

  The Firm has extensive experience in designing and developing high performance SQL/DS systems using development tools from IBM. Representative systems include a stock management system, registration management system for conference organizers, and automation of collecting processes for a South American government office.

- Database design

  The Firm continues to lead in the definition of methodologies for the design and performance evaluation of very large databases. We are recognized for our practical controllable methodologies which result in the timely design of databases, successful implementation of user requirements, and achievement of defined performance targets. We are also recognized for the definition and use of analytical and simulation-based approaches which predict database performance in most DBMS environments.

- Database conversion

  The Firm recognized early on that SQL/DS is a DBMS of choice for implementing medium size commercial transaction processing systems. We have assisted numerous clients across a broad variety of industries in their first implementation of relational database systems from most other non-relational DBMS's, including DL/I.

ANDERSEN CONSULTING
ARTHUR ANDERSEN & CO., S.C.

- Software productivity tools

  Based on our extensive experience in the development of systems, we identified, funded and developed our own proprietary systems development facilities to enhance the productivity of applications developers. These facilities include planning, analysis, design, code generation, test data management, data and database administration that make development more productive. These tools have seen extensive use in the delivery of application systems. In addition, these facilities are platform independent and as a result work in a variety of online environments. Depending on corporate preferences, mainframe applications may be developed on mainframes or on workstations for transfer to the mainframe after unit testing is complete.

- Hardware resources in development centers

  One of the most challenging aspects of conversion to a new DBMS is finding the machine time to convert existing data. Andersen Consulting can assist in this challenge through our development centers that can make available all the machine resources needed for the implementation of new systems and the associated conversion of databases. The centers are available through the facilities of our world-wide network AANET.

- Offshore development capability

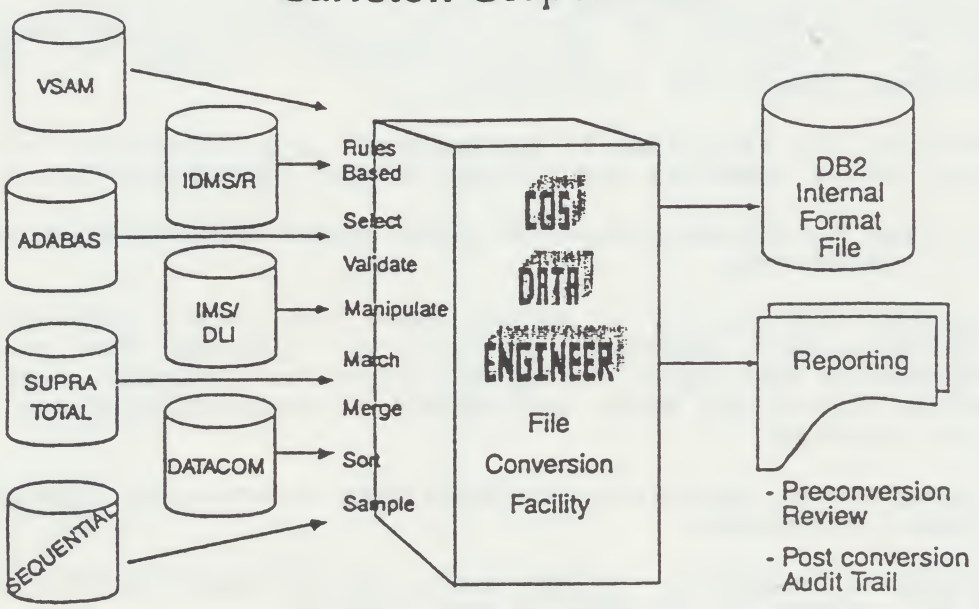
  Andersen Consulting was a leader in recognizing the practicality and utility of remote development of applications. We have created an infrastructure and defined methods so that our clients have access to programming resources. In turn, this provides highly cost effective rates to assure that development can be done at the lowest cost. In addition, we are a world-wide organization with consulting groups in over 45 countries. As a result, we are able to provide the resources to coordinate conversion efforts on a world-wide basis.

We are an IBM Business Partner

For more information, contact Consulting Information Services, Andersen Consulting, 69 West Washington Street, Chicago, Illinois 60602. Telephone (312) 507-6690, or your local Andersen Consulting office.

CARLETON CORPORATION

Carleton Corporation



COMPANY BACKGROUND

Carleton Corporation is a leading designer, developer, and marketer of information retrieval, manipulation, conversion and coexistence products for IBM mainframes. Carleton has its headquarters in Burlington, Massachusetts and supports over 250 sites in North America and around the world. The Carleton products are offered worldwide through local subsidiaries or representatives.

OVERVIEW

CQS-DATA ENGINEER for VSE provides a Fourth Generation language interface to generate both simple and sophisticated COBOL programs. These programs provide the user the ability to retrieve information, convert data files, create comprehensive data extracts, migrate database contents to SQL/DS and develop comprehensive reports. All the above can take advantage of the integrated audit functions inherent in **CQS-DATA ENGINEER for VSE.**

CQS-DATA ENGINEER for VSE provides for retrieval of information from any combination of standard IBM file structures (ie. VSAM, BDAM, ISAM etc,) as well as DL1 and SQL/DS. The user has the ability to access up to 20 input files of any type, match these files based on both simple and sophisticated key criteria, merge or join the appropriate files and create logical records for processing.

Through the application of user defined rules, databases can be analyzed, scrubbed (cleaned) and verified, and the information can be reformatted to address various presentation requirements. Output can be generated in any combination of the following formats: Internal SQL/DS format, Sequential format, PC format and Report format. Up to 19 sequential and/or PC format files and up to 100 Internal SQL/DS format files and/or Reports, can be generated from one pass of the input files.

CQS-DATA ENGINEER for VSE provides the most comprehensive, cost effective facility for information migration, database coexistence and information presentation.

8 NEW ENGLAND EXECUTIVE PARK • BURLINGTON • MA 01803 • TEL. (617) 272-4310 • FAX. (617) 272-2910

Business Partner

BENEFITS

- In addition to supporting all standard IBM file formats, **CQS-DATA ENGINEER for VSE** provides a specialized database interface to one or multiple major database systems including DL1, SQL/DS, Adabas, Datacom, IDMS/R and Total/Supra.

- Provides the facilities to accommodate pre-conversion review of source files, and a complete audit trail of the file conversion process.

- Provides a sampling capability which allows statistically valid test files to be generated.

- Provides the ability to merge, sort and verify information from many different data sources, as well as allowing the creation of new information and fields during the migration process. Additionally, many format conversion tasks are handled automatically based on the output type selected.

- A coexistence capability for providing information from or between one or many databases.

- An immediate benefit to the users and applications that require access to information in the SQL/DS environment.

FUNCTIONAL DESCRIPTION

CQS-DATA ENGINEER for VSE is dictionary based. Data definitions are stored in the CQS dictionary. Information may be selectively accessed; manipulation criteria defined and processing rules implemented through the CQS Language. The language provides full computational logic and allows the user to read files (up to 20), match files (up to 20), merge files and generate up to 100 conversion files and reports from one pass of the source files and/or databases.

Each conversion file will be in SQL/DS internal format and represents the contents to one SQL/DS Table. As such, the contents of up to 100 SQL/DS tables can be generated in each CQS program with only one pass of the input data. The reporting capabilities allow the user to review information prior to conversion and provide a hard copy or sequential file audit trail of the conversion process. Once the CQS program is complete, it is submitted to system in batch mode and a COBOL program is generated. The generator accesses the dictionary for all file information and embeds the appropriate I/O or CALL statements based on the dictionary entries. This generated COBOL program may be processed through an optimizer and stored in a standard production load library.

CQS-DATA ENGINEER for VSE is one of several Carleton products, each of which provide high quality productivity tools for today's data processing departments and information centers. Carleton Software products allow the integration of corporate databases, the information center, and personal computers and have been designed for use by both programmers and end-users.

For further information about **CQS-DATA ENGINEER for VSE** and other CQS products, please contact Carleton Corporation at (617) 272-4310, Fax number (617) 272-2910 or write to Carleton Corporation, 8 New England Executive Park, Burlington, MA 01803.

CTG

COMPUTER TASK GROUP INC.

Data Base Consulting Services
7650 Courtney Campbell Causeway
Suite 1220
Tampa, Florida 33607
(813) 289-4471
Fax (813) 289-6737

SQL/DS MIGRATION SERVICES

Facts About Computer Task Group

Computer Task Group (CTG) is an international consulting, systems integration, and professional services firm. We plan, design, build, implement, maintain, and migrate information, automation, and communications systems for major companies around the world. Full time professionals provide our services through an international network of more than 65 offices and application development centers in the United States, Canada, and Europe.

Founded in 1966, CTG has a track record of consistent growth and profitability. We are a public company traded on the New York Stock Exchange under the symbol TSK. Our 1990 revenues exceeded $243 million, with expectations of more than $280 million in 1991. We pride ourselves in the standards we maintain, the quality of services we deliver, and the diversity of solutions we provide.

CTG offers a total solution to your migration needs through an integrated effort, combining expertise in many highly technical specialty areas. The following paragraphs describe our capabilities to provide database to database migration solutions.

Traditional Migration / Conversion Services

Traditionally, conversions have been limited to relocating applications from one hardware platform to another, or one operating system to another on the same hardware. The automated conversion capabilities available were modest so that only "simple" transitions were possible. More complex migrations had to be done manually. This resulted in manpower intensive projects which seemed to never end.

Over the past twenty years, Computer Task Group has utilized increasingly more sophisticated tools to automate the conversion process. We now have specialized tools to move between literally dozens of different hardware environments. These tools have allowed us to progress from simple language translations to ever more complex on-line and data base migrations.

DOS to MVS conversions are delivered by CTG using a revolutionary, highly automated, conversion methodology that allows completion of these formerly unmanageable projects in a fixed time of only six months. We have successfully completed more than eighteen of these major conversions since 1986.

We feel that migrating to a relational database management system is key to establishing an integrated enterprise data architecture. This architecture holds the promise of supporting rapidly changing information requirements, in turn providing improved competitive positioning for your organization. Achieving this strategic architecture probably requires migration from your existing environment.

CTG is in a unique position to offer a broad range of skills in the existing database environment, the target database environment (SQL/DS), together with extensive experience with automated migration tools. We acknowledge that migrations to the target environment may take many months, and that coexistence strategies are critical to overall success. CTG has specialists with knowledge and experience in the following database products: DL/I, ADABAS, DATACOM/DB, IDMS, IDS-II, INGRES, M204, and ORACLE. Our expertise in the target database environments includes all the IBM relational products (DB2, SQL/DS, SQL/400, OS/2 EE). Thus we have established an integrated team approach providing a comprehensive migration solution. Currently, we are extending our existing automated tools to encompass the components necessary for arriving at the desired enterprise database architecture. CTG has a close working relationship with Bachman Information Systems, Inc.

What Computer Task Group Offers

Skills integration of specialists from diverse environments, combined with project management experience is the key to a smooth conversion. CTG can provide a total solution for you to move from your current database management system to IBM's strategic System Application Architecture. We handle all phases of a conversion from planning and preparation through user acceptance, documentation, and training.

An IBM Business Partner

CTG has been an IBM Business Partner since 1987. Jointly, we have successfully enhanced strategic business systems for organizations large and small.

For More Information

Please contact:
Computer Task Group
Database Consulting Services
7650 Courtney Campbell Causeway
Suite 1220
Tampa, Florida 33607
(800) 827-3267

General Electric Consulting Services Corporation
401 North Washington St., Rockville, MD 20850

Flexible Solutions From A Name You Know...

If you're ready to take advantage of the power of a relational technology such as IBM's SQL/DS, GE Consulting Services can help you get the job done.

At GE Consulting Services, we'll help you take advantage of new technology to meet the specific information systems needs of your organization. We know that each business has individual requirements that set it apart from all the others – and we create flexible solutions – solutions tailored to your needs – to help you gain the competitive edge.

Our Approach to Relational Technology

Increased user productivity. CASE technology. Increased processing flexibility. Enhanced ad hoc reporting. These are some of the reasons why businesses are considering relational technology today. And that's what SQL/DS delivers.

We can implement a new SQL/DS database with a minimum of impact on your existing DL1 source code by utilizing a "line-for-line" approach. Or we can implement a new SQL/DS database by applying a structured data modelling and redesign approach. We tailor our approach to your needs – the requirements of your business and users. We can help you decide which approach is best for your business – a solution that fits today's needs – and tomorrow's business.

People and Proven Expertise

GE Consulting has the people and proven expertise to make your information system work for you, to get you where you need to be, efficiently and cost-effectively. Our Application Engineering and Migration Services deliver more than program conversions – we design solutions that make intelligent use of your new technology investment from day one.

Over 1,100 of our technical specialists, located in 22 cities across the country, can provide you with strong, responsive, reliable support – backed by a unique network of national resources. Call our Application Engineering and Migration Services Group at **212-836-2362** for more information.

If you're looking for creative, flexible, business-oriented, cost-effective solutions tailored to your special requirements – count on support from a name you can trust. Count on GE Consulting.

Glossary of Terms

General Definitions

Term	Definition
Application (sometimes 'Application system' or 'Application suite')	A set of records and programs that support a specific business requirement (high level business function). Examples are; accounts receivable, payroll, demand deposit, etc. The same definition applies to DL/I and SQL/DS systems.
DDL	Data Definition Language. In SQL/DS, SQL; in DL/I, DBDGEN, PSBGEN and ACBGEN.
DML	Data Manipulation Language. In SQL/DS, SQL; in DL/I, DL/I or Data Language/One
Programs	One or more modules that perform some application function. Some of the modules may be shared with other applications, one example being error handling modules. An application program is a program that is used in the execution of an application as opposed to a systems program or utility program.

SQL/DS Terms and DL/I Correspondence

SQL/DS Term	Definition	DL/I Term	Comments
Catalog	The active, integrated location of physical and logical data definitions.	**DBD, PSB and ACB Librarles, DB/DC Data Dictionary**	There is no exact analogy between the SQL/DS Catalog and the DL/I data definition libraries. The Catalog consists of relational tables and can be queried with SQL. SQL is used to define entities in the SQL/DS Catalog. A special macro language is used to define DBDs and PSBs. The ACB is a run-time library containing DBD and PSB information. The DB/DC Data Dictionary is a separate, optional product in which many DL/I users store definitions of not only data but related objects as well.
Clusterlng Index	A table index that will aid SQL/DS in attempting to maintain the rows of the table in a physical sequence equivalent to the key of the index. A performance option.	**Sequence field, Physical insert rule, Processing sequence (PROCSEQ)**	There is no direct equivalence in that SQL/DS does not ensure retrieval in key sequence unless the SQL statement specifies ordering.
Column	An attribute of an entity. A data element in a relational table that is the smallest unit of data retrieval or modification.	**Field**	A portion of an DL/I segment. Note that not all fields must be defined when the segment is defined.

SQL/DS Term	Definition	DL/I Term	Comments
Database	An administrative unit encompassing a set of tables and related objects.	**Database**	Not a direct equivalent. An SQL/DS database is a collection of independent or interrelated pieces of information stored together to serve one or more applications. VM users can access multiple databases from an application program.
DBSPACE	A physical space defined to hold one or more tables, usually one.	**Database dataset, or dataset group (DSGROUP)**	Not a direct equivalent in that a database dataset normally holds data from multiple DL/I segment types.
Dependent Row	A row that references another row by means of a foreign key.	**Child Segment Occurrence**	In DL/I relationships between segments (parent-child, child-child) are via physical pointers or by physical juxtaposition of segments.
Dependent Table	A table that references another table by means of a foreign key.	**Child Segment Type**	In DL/I relationships between segments (parent-child, child-child) are via physical pointers or by physical juxtaposition of segments.
Foreign Key	Column containing the value of a primary key of a row in the same or another table for the purpose of identifying a relationship. SQL/DS uses the foreign key definition for the specification of referential integrity constraints.	**Parent pointer**	Every DL/I Child Segment occurrence must have a parent segment.
Package	The compiled set of interfaces required for the execution of a program. Includes mapping and navigation to the data.	**Program Specification Block (PSB)**	The PSB contains mapping and security but not navigation (except in the case of secondary indexing, where a processing sequence (PROCSEQ) may be specified).
Parent Row	A row that is referenced by a foreign key of another row.	**Parent Segment Occurrence**	In DL/I relationships between segments (parent-child, child-child) are via physical pointers or by physical juxtaposition of segments.
Parent Table	A table that is referenced by a foreign key of another table. May be the same or different table.	**Parent Segment Type**	In DL/I relationships between segments (parent-child, child-child) are via physical pointers or by physical juxtaposition of segments.
Primary Key	The unique identifier of a row.	**Segment Sequence Field**	DL/I segments may or may not have a Segment Sequence Field which may be unique or non-unique.

SQL/DS Term	Definition	DL/I Term	Comments
Referential Integrity	The specification of business constraints to the data base manager concerning the maintenance of relationships between tables. Implemented through primary and foreign keys.	**Parent-Child Relationship**	Every DL/I child segment must have one physical parent. In the case of Logical Relationships, a child segment may additionally have one logical parent for which there are rules governing segment insertion, replacement, and deletion.
Row	A set of attributes associated with an entity. One occurrence of the collection of columns representing the relational table.	**Segment Occurrence**	The smallest unit of retrieval or modification in a DL/I database.
SQL	Structured Query Language. ANSI and ISO standard language for relational data base.	**DL/I (Data Language/One)**	DML for retrieval, insertion, replacement, and deletion of DL/I segments.
Table	A set of rows with a common definition.	**Segment (or Segment Type)**	The collection of related fields which define the segment.
View	The ability to define a virtual table made up of data from one or more tables. May be dynamic or static.	**Program Communications Block (PCB) or Logical DBD**	A PCB within a PSB (q.v.) defines the (subset of) segments and/or fields which that program may process from the specified DBD. A Logical DBD provides an alternative view or inversion of segments which are usually from more than one logically related Physical DBD.

DL/I Terms and SQL/DS Correspondence

DL/I Term	Definition	SQL/DS Term	Comments
Child Segment	A DL/I segment which has a parent. All segments in a DL/1 database below the root segment are child segments. All child segments of a particular type may be linked together and linked to the parent segment.	**Dependent Table**	SQL/DS dependent tables are equivalent to DL/I child segments. Foreign key to primary key relationships replace DL/I linkages.
Database	A DL/I database is the collection of all segment types related to a single root segment type. DL/I can have any number of data bases under control of a single DL/I DBMS.	**Database**	All the tables and the DBMS itself in SQL/DS are considered a database. A single SQL/DS DBMS can have only one Database.
Database Dataset or Dataset Group	Physical space allocation for one or more segment types in the same database.	**DBSPACE**	Typically should contain only one table but may contain multiple tables.
DL/I (Data Language/One)	DL/I language for data retrieval and manipulation.	**SQL**	Structured Query Language. ANSI and ISO standard language for relational data base.

DL/I Term	Definition	SQL/DS Term	Comments
Field	A portion of a segment that identifies, or provides the value of an attribute of, the entity represented by the segment.	**Column**	An attribute of the table. The smallest unit of data transfer in SQL/DS is a column. The smallest unit of data transfer in DL/I is the segment.
HDAM Sequence Field	A root segment key field that is hashed (randomized) to a Relative Byte Address (RBA) for retrieval or storage.	**Primary Key**	Not necessarily a direct correlation. The HDAM sequence field will normally convert to a primary key; the converse is not true.
DL/I Log, Log Dataset(s)	System file containing recovery information. Inserts, deletes, replaces, and other events result in information being written to the DL/I Log. The Log is also used for accounting and statistical information and transaction history.	**SQL/DS Recovery Log**	Unlike the DL/I Log, the SQL/DS Log contains only information needed for data and unit-of-work recovery. SQL/DS has a security audit trace facility to monitor access to resources and the usage pattern of tables.
Logical Child Segment	A segment which has two parents, a physical parent and a logical parent. The purpose is often to relate the parents in a many to many relationship.	**Association Table**	The same function is performed in SQL/DS by a table which is the dependent of two (or more) parent tables. The row contains a foreign key for each parent table.
Logical Database	A set of segments from one or more logically related physical DL/I databases.	**Referentially related tables**	There is no exact correspondence in SQL/DS. The closest is a group of referentially related tables.
Occurrence (Segment)	An instance of a segment.	**Row**	An instance of all the columns in a table.
Parent	The parent segment of a parent-child relationship.	**Parent**	The parent table in SQL/DS controls a set of dependent tables using primary and foreign keys instead of linkages (pointers) for the relationship.
Parent-Child Relationship	An DL/I parent-child relationship is an association between segments. Generically, the association is one to many with the "one" being a parent and the "many" being a child. The relationship between parent and child is maintained with links or pointers or segment physical juxtaposition.	**Parent table, Dependent table**	The parent/dependent definitions of relational data base are closely equivalent to the DL/I parent-child relationship. Referential constraints are specified in both cases. Relational maintains the relationship through the use of primary and foreign keys.

DL/I Term	Definition	SQL/DS Term	Comments
Primary Index	Root segments are retrieved in sequence by a primary index if the DL/I database organization is HISAM or HIDAM.	**Clustering Index**	Not an exact analogy. SQL/DS will attempt to maintain data in a physical order corresponding to the logical sequence according to the values in the indexed column(s). Exceptions may occur later as rows are inserted and deleted. To guarantee logical sequencing, the SQL ORDER BY clause must be used. The SQL/DS optimizer decides whether the clustering (or any other) index would be beneficial for retrieval.
Program Specification Block (PSB)	The specification of all of the data (segments) that a program may access. Includes security constraints and access restrictions. May be one per program or could be used by more than one program.	**SQL/DS Views, Package**	There is not a direct correlation, but the PCBs (q.v.) within a PSB provide capabilities similar to a set of SQL/DS views which are used by a Package.
RBA (Relative Byte Address)	The physical storage location of a segment. May be used by DL/I-DB for retrieval of a segment occurrence. Provided by a program (i.e. a randomizing module) for HDAM database organization.	**None**	There is no SQL/DS equivalent function that the user has any control over.
Sequence Field	A field whose values determine the physical sequencing, or sequence of appearance on a pointer chain, for a given segment type. It may be assumed that the order of retrieval is according to the values of the sequence field(s).	**Indexed Column**	Not an exact analogy. SQL/DS will attempt to maintain physical sequence according to the values of the indexed column(s) if the indexing is clustering. Other indexes are simply a navigation aid which may or may not be selected by the Optimizer depending on the situation. In any case, there is no assurance that the application will retrieve the data in that sequence unless an ORDER BY is specified in the SQL request. If an index is used in support of an ORDER BY or search condition, performance is often greatly improved.
Segment	A defined collection of data elements (fields) that represents a physical entity in the database.	**Table**	A defined collection of columns that represent the table.

Index

A

ACT 27, 44
Ad-hoc query 8
Aggregations 88
Analysis
  Column 48
  Complex program 50
  Database design 45
  Estimating 49
  Field 48
  Foreign key 47
  Primary key 47
  Program 50
  Repeating fields 48
  Rules of thumb 46, 49
Analysis of DL/I calls 93
Analysis of DL/I commands 93
Analyzing existing DL/I application programs 86
ANSI 4
Application 25, 26, 193
  Analysis 25
  Inventory 25
Application analyst 18
Application extension 8
Application program migration 85
Application programmer 18
Application selection 20
Authorization 75

B

Bachman Information Systems 22
Backup 168
Batch 169
BGS Systems 22
Built-in functions 56, 88
Business analyst 18

C

Carleton Corp. 22
CASE tools 92
Catalog 193
Changes, Freezing 19
Changes, Functional during migration 19
Child segment 195, 196
CICS 43, 44, 50, 74, 169
  ACT 44
  PPT 27, 43
CICS Playback 22
Clustering Index 193
COBOL programs 134
Coexistence 161—165
  Extract 161
Coexistence *(continued)*
  Programs updating one DBMS 162
  Programs updating two DBMSs 162
Coexistence approach 54
Column 36, 48, 56, 193, 196
  See also Element, Field
Column selection vs. segment selection 131
Combining segments 80
Command codes, DL/I 125
Common modules 92
Complex program 50
Composite fields 56
Compuware 22
Concurrency 74
Consultants 16
Converting DL/I calls 89
Converting DL/I commands 89
CQS-Data Engineer 22
Creation of SQL/DS objects 82
Cross System Product 6, 21, 22
CSP See Cross System Product
Cursor 123
Cursor-controlled operations 123
Cutover 141
CV 169

D

Data coexistence 78
Data conversion
Data conversion validation 84
Data dictionary 61
Data extract 77
Data migration 20
Data modification commands 117
Data naming considerations 57
Data related between DL/I and SQL/DS 71
Data security 71
Data volume 81
Database 194
  Design 22, 53
  Design analysis 45
  Inventory 27
Database dataset 195
Database design 20
Database design checklist 75
Database design philosophy 55
Database design techniques 53
  Coexistence approach 54
  Design information sources 55
  Limited redesign approach 55
  One-for-one approach 53
  Redesign approach 53

Databases 73
DATALOAD 77, 78, 79, 83
Date fields 83, 121, 128
Date handling 90
DBA
  DL/I 17
  SQL/DS 18
DBD 27, 33
DBEDIT 22
DBEDIT PLUS 23
DBSPACE 194, 195
DBSPACEs 73
DBSU 77, 78, 79
DB/CENTER 23
DB/EDITOR 23
DB/REORGANIZER 23
DB/REPORTER 23
Decision support 8, 161
DECLARE CURSOR statement 107
Defaults 136
Delete rules 67
Delete (DLET) call 117
Delete (DLET) command 117
Dependent 194, 195
Dependent segments 71
Design
  Database 22, 53
  Extensibility 5
  Productivity 5
Design information sources 55
Designing for relational 61
Distributed data 6, 7
DL/I
  ACT 27
  Batch 169
  Child segment 195
  Database dataset 195
  DBA 17
  Field 48
  HDAM Sequence Field 196
  Inventory 27
  MPS 43, 44
  MPS batch 169
  Parent segment 196
  Relationship 196
  Repeating fields 48
  Utilities 169
DL/I database unload 77
DRDA 7

E

Education 20, 21
Element 196
  See also Column, Field
End-user access 8
End-user support 8
Environment 20
Error handling 89, 131
ESA 5, 6
Estimating 49
EXPLAIN 76, 143
EXPLORE 23
Extensibility 5
Extract 161
Extract Source 78
Extraction program 77

F

Field 27, 35, 48, 56, 196
  See also Column
  Length 31, 32, 36
  Repeating 36
    See also Repeating fields
  Sequence field 35
  Type 31, 32, 36
  Virtual field 32
Field initialization 90
Field-level programming 90, 136
Field-level sensitivity 56
Filler fields 56, 130
FIPS 4
Foreign key 47, 57, 59, 81, 194, 195, 196
  Building 77
  Construction 81
  Determination 69
Freezing of changes 19
Functional changes 19

G

Get Hold Next (GHN) 105
Get Hold Unique (GHU) 95
Get Next In Parent command 109
Get Next within Parent (GNP) 109
Get Next (GN) 105
Get Unique (GU) 95, 100
GHNP call 109
GNP command 109
GO processing option 125
Goal Systems International, Inc. 23
GPR 23
Group level items 70
Guest sharing 6

H

HDAM 74, 196
Hierarchical structure 59
Hierarchical to relational considerations 58
Host variables 92, 134
Host variables, structures 131

I

IBM SQL Master/VM 22
Implementation 20
Incompatible formats 81
Index database 33
Indexes, SQL/DS 74
Indicator variables 136
Initialization of fields 136
Input/output
Insert (ISRT) call 120
Insert (ISRT) command 120
Integration test 20
Integrity 3, 7, 52
  See also Referential integrity
Intermediate extraction files 78
Internal mapping 81
Inventory 20
  Application 25
  Columns 39, 41
  Database 27
  DBD 33
  DL/I exit routine 31
  Exit routine 32
  Fields 35, 41
  Number of DL/I calls 43
  PCB 29
  Program description 43
  Program size 43
  Programs 42, 44
  PSB 29, 44
  Segments 33, 38
  SENFLD 31
  SENSEG 30
  Tables 38
  VIRFLD 32
ISO 4
ISQL 92
Iterative processing 125

J

JCL 44, 169
  DLBL 170
  EXTENT 170
  FILEDEF 170
Job control
  See JCL
Join 4, 7
Justification 3
  Relational 3

K

Key
  See also Foreign key
  See also Primary key
  Foreign 194
Key changes 124
Key construction 80
Key feedback area 62

L

Limited redesign approach 55
Loading data 77
Loading tables 83
Locking 125
Log 196
Logical child segment 34
Logical database 33, 196
Logical database design 60
Logical relationship rules 124
Logical relationships 68
Logical unload utilities 79

M

Methodology 141
Migrating data 77
Migration assistance software 22
Migration considerations 11
Migration estimates 48
Migration of dependent segments 71
Migration of root segments 72
Migration personnel 16
Migration programs 51
Migration skills 17
MPS 43, 44
MPS batch 169

N

Naming conventions 92
Navigation and position 60
Navigation commands 95
Normalization 3, 55, 56, 62
NULL 40
Number of DL/I calls 43

O

Occurrence 196
OCCURS 36, 48
OCCURS DEPENDING 48
One-for-one approach 53
Operating environments 6
Operations 20, 167
Operations personnel 19
Optimizer 5
Order 59
ORDER BY statement 107
Outer join 113
Overlapping fields 63

P

Package 194
Parent 194
Parent segment 196
Path call 111
PCB 29
Performance 141
Personnel 16
Physical child segment 34
Physical database 33
Physical database design 72
Physical table definition 72
Planning 20
Pointers 60, 62
Positioning 60, 123
Positioning calls 86
Positioning vs. cursor-controlled operations 123
Post project review 20
PPT 27, 43
Primary key 47, 57, 59, 80, 194, 195, 196
  Building 77
  Construction 80
  Determination 69
  Update of 124
Processing option GO 125
Processor requirements 21
PROCSEQ 30
Production cutover 20
Program 50
  Complex 50
Program description 43
Program execution 91
Program inventory 42
Program migration 20, 94
Program migration checklist 138
Program migration strategy 91
Program preparation 91
Program size 43
Program testing 91
Program type 43
Programmer productivity 5
Programming language 26, 29, 43
Programs 42, 193
  Description 43
  Inventory 42
  Number of DL/I calls 43
  PSB 44
  Size 43
  Type 43
Programs to be migrated 86
Project - relational operation 7
Project analysis
  Column 48
  Database design 45
  Field 48
  Foreign key 47
  Primary key 47
  Repeating fields 48
Project plan 20
Project planner/manager 17
PSB 27, 29, 44, 197
PSB replacement 74
Pseudo-conversational transactions 74

Q

QMF 92
QMF report or query, to replace program 86
Query, ad-hoc 8

R

Record 197
Recovery 168
Redefined fields 63
REDEFINES 81
Redesign Approach 53
Redundant data 65
Referential integrity 4, 7, 29, 37, 59, 67, 77, 123, 125, 194, 195, 196
  Delete rules 67
Relational model 3
  Conceptual simplicity 3
  Industry direction 4
  Language 3
  Operations
    Join 7
    Project 7
    Select 7
    Set 7
Repeating fields 48, 62
Replace (REPL) call 119
Replace (REPL) command 119
Replacement of DL/I interface specifications 88
Replicated data 77
Review 143
Root segments 72
Row 195, 196
RPG 26
RTNAME 31, 32
Rules of thumb (estimates) 46, 49

S

SAA 6
SAA LanguageAccess 8
Sampling technique 82
Screens
Scrolling 92, 133
Search keys 90, 114
Secondary index 30, 74
Secondary indexes 64
Security 20, 71, 75, 153
Segment 27
  Concatenated key length 34
  Duplicate segments 34
  Inventory 33

Segment *(continued)*
Logical child 34
Parent segment 34
Physical child 34
Virtual logical child 34
Segment combination 66
Segment insert rules 70
Segment-to-table mapping 79
Segment/table format 59
Select - relational operation 7
SENFLD 27, 31
Exit routine 31
SENSEG 27, 30
Sequence field 62, 80
Sequencing 4
See also Order
Set 196
Set operation 7
Set processing 4, 125
Skills
Migration 17
Software requirements 22
Splitting segments 64, 79
SQL 3
SQLCIREO 173
SQL/DS
Application extension 8
Catalog 193
Clustering Index 193
Column 193
Columns 36
Database 194, 196
DBA 18
DBSPACE 194, 195
Decision support 8
Dependent 194
Distributed data 6
Distributed data. 7
End user access 8
End-user support 8
Foreign key 194
Optimizer 5
Parent 194
Plan 194
Primary key 194
Query 8
Referential Integrity 195
Row 195
SAA 6
Table 195
Technology support 5
View 195
VM/ESA utilization 5, 6
SQL/MENU 23
SQL/RADIUS 23
SSA modifications 126
Storage pools 73
System backup/recovery 20
System test 141
Systems Application Architecture 6
Systems programmer 18

T

Table 195, 197
Team leader 17
Technology support 5
Test 22
Test database extraction 82
Third normal form 56
Timestamps 121
TPNS 22

U

Union 4
Unit testing 138
Unload
Unload frequency 82
Unload utility file 81
Update of primary key 124
UPDATE STATISTICS 150, 173
Use of cursors 87
Utilities 170

V

Variable length segments 64
View 155, 195
Views 75
VIRFLD 32
Virtual field 32
Exit routine 32
Virtual logical child segment 34
VM 6
VM Application Planner 22
VM Solutions 23
VM Systems Group, Inc. 23
VM/ESA utilization 5
VSE 6
VSE/ESA utilization 6

W

WHERE clause 97, 104
WITT 22, 144, 179
Workload analysis 20

© IBM Corp. 1991
International Business Machines Corporation
Data Management Market Support
C16/010
5600 Cottle Road
San Jose, Calif. 95193

Printed in the United States of America
12/91
All Rights Reserved

References in this publication to IBM products or services do not imply that IBM intends to make them available outside the United States.

GH21-1084-00